Improving Mobile Infrastructure for Pervasive Personal Computing

Ajay Surie

CMU-CS-07-163

November 2007

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
M. Satyanarayanan, Chair
David A. Eckhardt

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research was supported by the National Science Foundation (NSF) under grant number CNS-0509004 and by the Army Research Office (ARO) through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Improving Mobile Infrastructure for Pervasive Personal Computing	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The emergence of pervasive computing systems such as Internet Suspend/Resume has facilitated ubiquitous access to a user's personalized computing environment by layering virtual machine technology on top of distributed storage. This usage model poses several new challenges, such as establishing trust in unmanaged hardware that a user may access, and efficiently migrating virtual machine (VM) state across low-bandwidth networks. This document describes Trust-Sniffer, a tool that reduces the security risks associated with transient use by helping a user to gain confidence in software on an untrusted machine. The root of trust is a small, user carried device such as a USB memory stick. Trust-Sniffer verifies the on-disk boot image of the target machine and incrementally expands the zone of trust by validating applications, including dynamically linked libraries, before they are executed. An application is validated by comparing its checksum to a list of known good checksums. If a binary cannot be validated, its execution is blocked. This staged approach to establishing confidence in an untrusted machine strikes a good balance between the needs of security and ease-of-use, and facilitates rapid transient use of hardware. This document also describes a solution to optimize the transfer of large amounts disk and memory state for VM migration, based on opportunistic replay of user actions. The term opportunistic means that replay need not be perfect to be useful. In contrast to other replay techniques, opportunistic replay captures user interactions with applications at the GUI level, resulting in very small replay logs that economize network utilization. Replay of user interactions on a VM at the migration target site can result in divergent VM state. Cryptographic hashing techniques are used to identify and transmit only the differences. I describe the implementation and associated challenges of a prototype system that supportsVMmigration, and present encouraging results with this prototype that show savings of up to 80.5% of bytes transferred.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**75**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: Virtual machines, mobile computing, nomadic computing, pervasive computing, transient use, establishing trust, untrusted terminal, load-time validation, trusted computing, Xen, VMM, wireless networks, compression, performance, log-based strategies, content-addressable storage, CAS, seamless mobility, Internet Suspend/Resume®, OpenISR®

To my family

Abstract

The emergence of pervasive computing systems such as Internet Suspend/ Resume has facilitated ubiquitous access to a user's personalized computing environment by layering virtual machine technology on top of distributed storage. This usage model poses several new challenges, such as establishing trust in unmanaged hardware that a user may access, and efficiently migrating virtual machine (VM) state across low-bandwidth networks.

This document describes Trust-Sniffer, a tool that reduces the security risks associated with transient use by helping a user to gain confidence in software on an untrusted machine. The root of trust is a small, user carried device such as a USB memory stick. Trust-Sniffer verifies the on-disk boot image of the target machine and incrementally expands the zone of trust by validating applications, including dynamically linked libraries, before they are executed. An application is validated by comparing its checksum to a list of known good checksums. If a binary cannot be validated, its execution is blocked. This staged approach to establishing confidence in an untrusted machine strikes a good balance between the needs of security and ease-of-use, and facilitates rapid transient use of hardware.

This document also describes a solution to optimize the transfer of large amounts disk and memory state for VM migration, based on *opportunistic replay* of user actions. The term opportunistic means that replay need not be perfect to be useful. In contrast to other replay techniques, opportunistic replay captures user interactions with applications at the GUI level, resulting in very small replay logs that economize network utilization. Replay of user interactions on a VM at the migration target site can result in divergent VM state. Cryptographic hashing techniques are used to identify and transmit only the differences. I describe the implementation and associated challenges of a prototype system that supports VM migration, and present encouraging results with this prototype that show savings of up to 80.5% of bytes transferred.

Acknowledgments

First, I would like to thank Satya. Satya has been a great advisor and mentor. He never hesitated to take time out of his busy schedule to meet with me, and could always be counted on for helpful feedback at the right time. When I felt frustrated or confused, he supplied appropriate doses of encouragement.

I'd like to thank Adrian Perrig, with whom it was a privilege to work with. He was very patient as I learned the fundamentals of computer security. Adrian constantly provided me with valuable feedback and guided me through the process of writing my first academic paper.

I am grateful to Dave Eckhardt for being on my committee and reading and reviewing the thesis at very short notice. He also provided me with many insights and suggestions during the course of my graduate study.

I'd like to acknowledge my colleagues from the University of Toronto, Andres Lagar-Cavilla and Eyal De Lara. It was a pleasure to work with them, and without their efforts my final project could not have been successful.

Working with the members of my research group, Niraj Tolia, Benjamin Gilbert, Jan Harkes, Adam Goode and Matt Toups has been a great experience. They are an extremely talented group of individuals, and always gave me helpful comments on my ideas. Jan was a great mentor and was very patient as I slowly acquired familiarity with Coda. Benjamin's assistance with implementation and the OpenISR codebase was invaluable. I'd also like to thank Tracy Farbacher for her assistance with coordinating meetings and making sure everything ran smoothly.

My good friends and colleagues, Adam Wolbach, Cinar Sahin, and Zhi Qiao were readily available when I needed help and motivated me when I struggled. Along with Alex Knecht, Azim Ali, David Fu, Hugh Dunn, Dana Irrer and Jeff Bourke, they provided much needed distractions at the appropriate times.

Finally, I'd like to thank my parents, Ricky and Gita Surie for emphasizing the impor-

tance of balancing work and play. Thanks to my sister Aditi for her encouragement and support.

Ajay Surie
Pittsburgh, Pennsylvania
November 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are a wide range of technologies available today that allow users ubiquitous access to their personalized computing environments. Thin client technologies such as *Terminal Services* [54] and *VNC* [46] allow remote access to a user's desktop. *MiGO* [4] allows a user to carry his personalized settings on a USB memory stick and apply these settings to a computer "on the fly." With *Internet Suspend/Resume*®, a user's personalized computing environment follows him through the Internet as he travels [50, 49, 31].

Pervasive computing systems like those described above have inspired new usage models in mobile computing, where *transient* use of PCs has become more common. This usage model presents many new challenges, such as establishing trust in unmanaged hardware that a user may access, and efficiently migrating state associated with the user's computing environment across low bandwidth networks. The focus of this document is on addressing these two challenges in the context of Internet Suspend/Resume (ISR).

This chapter begins with an overview of the ISR system. In Section 1.2, the two challenges are discussed in further detail. The next section describes the approach used to address these issues in ISR. The chapter concludes with a roadmap of the rest of the document.

1.1 Internet Suspend/Resume

In the emerging ISR model of mobile computing, users take advantage of pervasive deployments of inexpensive, mass-market PC hardware rather than carrying portable hardware. The driving vision of ISR is that the plummeting cost of hardware will someday

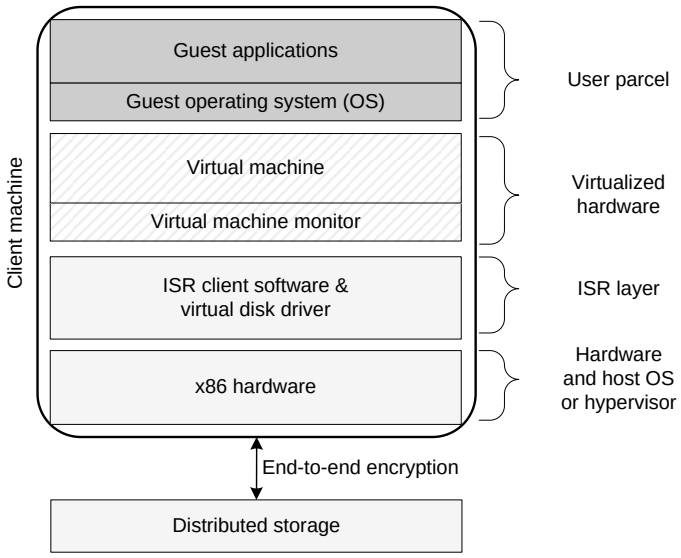

Figure 1.1: Modular Structure of an Internet Suspend/Resume Client

eliminate the need to carry one's computing environment in a portable computer. Instead, an exact replica of the last checkpointed state of a user's entire computing environment (including operating system, applications, user data files and customizations) will be delivered on demand over the Internet to hardware that is located near a user. Thus, ISR cuts the tight binding between PC state and PC hardware. Since all process execution and computation takes place on the hardware located nearby, ISR is able to provide the user with low-latency interactivity. ISR is implemented by layering a virtual machine (VM) on distributed storage. The architecture of an ISR client is depicted in Figure 1.1. The VM encapsulates the user's *parcel*, which is comprised of execution and user customization state. Distributed storage transports that state across space and time.

VM state consists of a virtual disk and a memory image, which is stored on centrally managed, well connected *content servers*. When a user resumes his computing session on a client machine, the user's parcel state is fetched from the content servers. The ISR model of distributed computing is depicted in Figure 1.2. To reduce the amount of data transferred between content servers and resume sites, ISR uses content addressability techniques. The disk image in a parcel is divided into a number of fixed sized *chunks*. When the user resumes his parcel at a new location, disk chunks are fetched over the network on demand as they are accessed. All data transferred over the network is compressed to improve effiency, and encrypted to protect user privacy. ISR maintains a local cache of all disk chunks as they are accessed, and a chunk only needs to be fetched over the network if

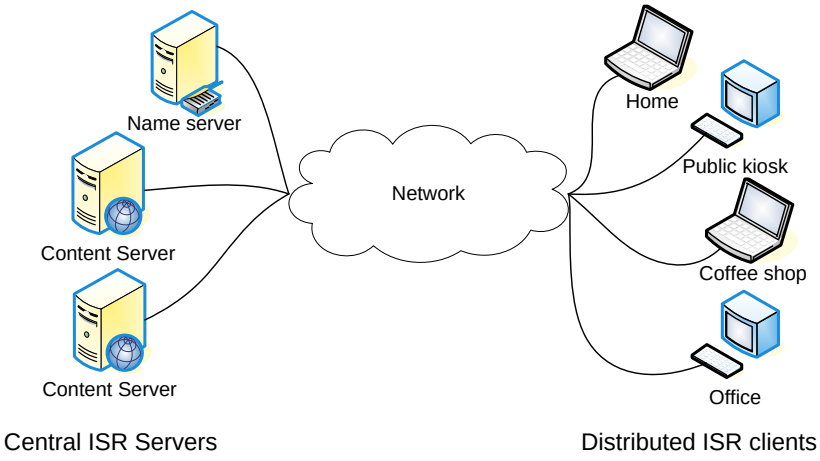

Figure 1.2: Distributed Computing with Internet Suspend/Resume

the latest version of the chunk does not already exist locally. When the user suspends his parcel, only modified chunks must be uploaded to the content server.

1.2 Two Challenges

Transient use of PCs involves a user accessing his or her computing environment for short periods of time from multiple locations, potentially with hardware that the user does not own or manage. In order for this usage model to become practical with ISR, two challenges must be addressed.

1.2.1 Rapidly Establishing Trust in Nearby Hardware

Establishing trust in computing platforms is becoming more important as users begin to access computing environments from multiple locations. Even for common day-to-day user tasks, such as web browsing, online shopping, checking email, or editing documents, the risk of a user's personal data being compromised by a malicious system is high. For example, at home, a shared computer could become compromised through a child's unwitting download of a virus over the Internet. Public computers such as those in a cluster, an Internet cafe or a hotel could easily be rendered unsafe by an attacker installing malware, such as a keystroke logger.

Today, when a user sits down to use a computer in his office or home, he implicitly

assumes that the machine has not been tampered with. This is a reasonable assumption today because physical access to the machine is restricted. The same assumption applies to a portable computer that is physically safeguarded by the user at all times. In contrast, physical access to unmanaged hardware is generally unrestricted, and unmanaged systems often suffer from problems such as poor maintainence, which increase security risks. If the ISR vision of transient use of hardware is to become commonplace, it is necessary for a user to be able to quickly establish a similar level of confidence in unmanaged hardware.

1.2.2 Efficiently Migrating Parcel State

With ISR, users may access their personalized computing environments from multiple different resume sites. For example, a common usage pattern is accessing one's computing environment from two computers, one at home and one at work. While some resume sites, such as the user's office may have plentiful bandwidth, residential and public resume sites are often bandwidth constrained.

ISR's encapsulation of user state in a VM simplifies simplifies many aspects of moving computation from one site to another. However, this simplification comes at a price: VM state is typically large, often tens of GB in size. At low bandwidths, the transfer time for such a large amount of parcel state is excessive. For ISR use to be seamless across multiple resume sites and varying bandwidth conditions, migration of state across low-bandwidth links needs to be more efficient.

1.3 The Thesis

The goal of this research is to improve the underlying mobile infrastructural mechanisms in Internet Suspend/Resume that support transient use of PCs. While the techniques employed to improve security and state migration do not guarantee a minimum level of performance, the user is better off than if no improvements are used. The thesis statement is thus:

> *Pervasive personal computing can be made more secure by enhancing a user's ability to make trust decisions in his local computing environment. Performance can be significantly improved by opportunistically associating user interactions with generated state changes. These accelerations can be implemented without source code modifications to major software components, such as operating systems or virtual machine monitors.*

1.3.1 Scope of Thesis

This thesis focuses on improving transient use of PCs across varying conditions of security and connectivity in the local computing environment. This thesis makes the following important assumptions:

- The ISR client software needed to access a user's parcel at a resume site, encompassing the user's *trust footprint*, is small. In addition, no user relevant state is accessed or stored outside a parcel.
- Available hardware at resume sites is either free from malicious tampering or tamper evident. Software at the resume site may be compromised.
- ISR content servers are strongly connected to the network, and have ample storage and compute resources. Clients may be resource limited and/or connected via bandwidth constrained links to the wide area network.
- Optimizations for parcel state migration address ISR usage at frequently visited resume sites, which hold previous checkpoints of parcel state.

1.3.2 Approach

The overall approach of this work is best characterized as *user-influenced optimization*. In other words, the system leverages particular usage patterns and tendencies of mobile users as hints to enhance security and performance. While the system does not provide a minimal guarantee of performance, it strives to adapt when conditions are more favorable than expected. The work presented here is based on the following guiding principles:

- *Don't make performance worse than it is now.*
  While the use of virtualization technology precludes achieving native performance, the system should not further degrade performance from current levels.
- *Do things in the background when possible.*
  As computing systems have evolved, the most constrained resource has become user attention. This strategy minimizes interruptions to a user's workflow by performing as much work in the background as possible.
- *Do focus on simplicity of operation and minimize burdens placed on the user.*
  Systems with complex usage models often resist adoption. Users are more likley to

embrace functionality that is either built into the system or requires little effort to understand.

1.3.3 Validation of Thesis

This thesis was investigated by prototyping the strategies described previously within the context of ISR. ISR has been developed since 2001, and hence provides a large body of existing experience as well as a stable framework in which to evaluate these ideas. Empirical measurements and controlled experiments are used to validate the thesis statement.

1.4 Document Roadmap

The rest of this document is divided into four chapters. Chapter 2 describes techniques to rapidly establish trust in hardware intended for use as an ISR resume site. Chapter 3 describes a mechanism to improve migration of VM state between ISR resume sites and content servers. Empirical results are presented at the end of each chapter. Chapter 4 discusses work related to this thesis. The document concludes with a discussion of the contributions of this thesis and future work in Chapter 5.

Chapter 2

Rapid Trust Establishment

Most existing solutions in the literature to enhance security rely on system administrators to undertake the task of protecting users from attacks. In addition, they do not address the concerns associated with mobile users accessing personalized computing environments from multiple locations. This chapter describes the creation of a tool called *Trust-Sniffer* to enhance security for mobile users. Using Trust-Sniffer to explore new portions of the design space, the research focus is an attempt to improve security as much as possible while reducing the requirements placed on the user, and preserving ease of use.

Trust-Sniffer helps a user incrementally gain trust in an initially untrusted machine. The root of trust in Trust-Sniffer is a small, lightweight device such as a USB storage device that is owned by the user and carried by him at all times. This *trust initiator* device is used to boot the untrusted machine so that Trust-Sniffer can examine its local disk and verify the integrity of all software that would be used in a normal boot process. Once the integrity of the normal boot process is verified, a reboot from disk is performed. In this step, the *trust extender* module of Trust-Sniffer is dynamically loaded into the kernel. As its name implies, this module is responsible for extending the zone of trust as execution proceeds. On the first attempt to execute any code that lies outside the current zone of trust (including dynamically linked libraries), the kernel triggers a *trust fault.* To handle a trust fault, the trust extender verifies the integrity of the suspect module. Execution stops if the module's integrity cannot be established.

2.1 Background

This section gives an overview of the methods used by attackers to compromise vulnerable systems. It then provides background information about trusted computing and introduces existing work in the literature that is relevant to the system.

2.1.1 Methods of Intrusion and Code Modification

Computers connected to a network are susceptible to remote attacks where a malicious party exploits a software vulnerability to obtain access to the system. Once the attacker obtains access to the system, he can install a kernel-level rootkit. The rootkit could allow the attacker to maintain unauthorized system access by replacing system binaries with malicious ones. It also allows the attacker to execute malicious code undetected by the operating system. Other threats include viruses, worms and malware.

These problems have been commonly studied in the context of hardware managed by administrators, and are an even bigger threat to systems maintained by non-expert users. The transient usage model also complicates the intrusion problem, because users typically have physical access to the system. This allows certain attacks to be executed even without network connectivity.

Modifications to the software stack could cause potentially disastrous results, especially when users begin to trust their personal data to unmanaged systems. Such modifications could be as innocuous as a commonly used system utility, such as `ps`, or as significant as modifying the operating system, such as with a Loadable Kernel Module (LKM) [27]. Even without administrative privileges to the system, BIOS permitting, an attacker could boot a shared system with a live CD such as Knoppix [2] and carry out unauthorized modifications to the software on the local hard disk drive. Sensitive data of subsequent users could be compromised by a malicious application that the attacker installed.

2.1.2 Trusted Computing Primitives

Most solutions that protect systems from intrusions involve mechanisms that ensure that the integrity of a system is maintained in the face of attacks. This is accomplished by detecting changes to the software stack on the system and comparing them to a known baseline configuration.

Two well known techniques that follow this strategy are *trusted boot* and *secure boot*.

However, both require modifications to the platform. The Trusted Computing Group's (TCG) standards specify the use of a secure co-processor, the Trusted Platform Module (TPM) to store sensitive state [8]. Once data is stored in the TPM, it cannot be tampered with. The *trusted boot* mechanism makes use of the TPM to establish confidence in a PC's bootstrap process. It extends a chain of trust from an established unmodifiable base called the *root of trust*. When a machine boots, the BIOS verifies the bootloader by computing a SHA-1 hash of its executable code, and saves this information to the TPM. Following this, each component in the boot process verifies the subsequent component that is loaded and extends the TPM with this information. A remote party can then verify the integrity of the boot process using a challenge response mechanism to obtain the state stored in the TPM.

The *secure boot* approach starts with a root of trust which is the initial BIOS bootstrap code. Before loading each subsequent piece of software, the current component verifies the digital signature of the next component [17]. If the digital signature of a component is incorrect, it is prevented from executing and the boot process is halted. The secure boot procedure does not make use of any special hardware, but it requires modifications to boot components, notably the BIOS, which needs to contain signature verification code. The significant distinction between the two approaches is that with trusted boot, the TPM only provides *post-facto* discovery of anomalies in the software stack, whereas secure boot protects the system from malicious code by preventing its execution. In addition, although secure boot prevents the execution of malicious code, it does not provide a way for a third party to verify the code running on the system.

2.2 Design Overview

The goal of Trust-Sniffer is to enhance security with modest user effort. A key design principle of Trust-Sniffer is to validate only the software needed for a user's task. This design principle is motivated by unique characteristics of the ISR usage model. Specifically, most of a user's execution environment is fetched from a trusted server over an authenticated and encrypted channel. This includes the guest operating system and applications that execute inside the user's VM. Cached VM state on disk is always encrypted, and its integrity is verified by ISR before use by a VM. Thus, it is necessary to verify the integrity of only a small core of local ISR and Linux software. Other compromised state is not a threat if it is never used when a machine is functioning as an ISR client. This minimalistic approach speeds the process of trust establishment and broadens the ubiquity of ISR. A second important design principle is to prevent the execution of untrusted software. This is in contrast to attestation techniques, which facilitate the detection of untrusted software

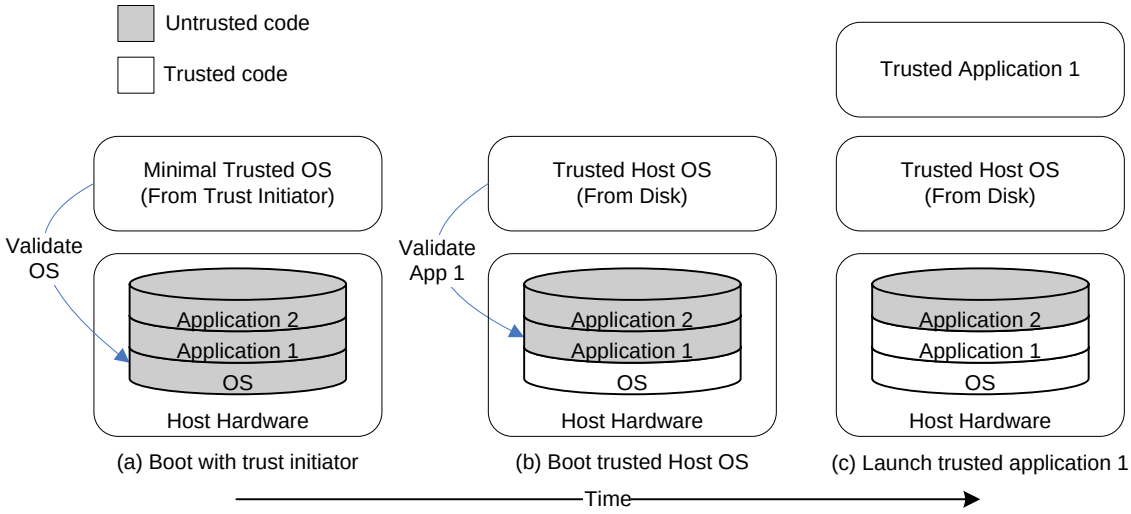

Figure 2.1: Trust-Sniffer's staged approach to trust establishment. Initially all software is untrusted. (a) The trust initiator's boot uses a minimal trusted OS to validate the on-disk OS. (b) Next, the trusted host OS is booted from disk, which validates applications as required. (c) The host OS permits only trusted applications to execute.

but do not prevent its execution.

2.2.1 Staged Approach

As shown in Figure 2.1, Trust-Sniffer uses a staged approach to trust establishment:

- *Establishing a root of trust*
  The user first performs a minimal boot of the untrusted machine from a trust initiator device. From a user perspective, this device has two major advantages: (i) its small size and passive nature make it inexpensive, easy to replace and convenient to carry around; and (ii) relying on a trusted physical possession to establish trust is convenient and easy to understand. This stage involves a minimal boot, because its sole purpose is to verify that it is safe to perform a normal boot using the on-disk operating system and associated boot software. The minimal boot ignores the network and devices other than the disk.

- *Booting the on-disk operating system*
  The second stage performs a normal reboot, which is now known to be safe. This

process ensures that a full suite of drivers is obtained for the local hardware (such as graphics accelerators) as well as correct local environment settings such as those for printers, networks, and time. As part of this boot process, Trust-Sniffer loads the kernel module needed to handle trust faults.

- *Validating other local software on trust faults*
  As each component of ISR client software is accessed for the first time, trust faults are generated and Trust-Sniffer validates the component's integrity. An advantage of *on-demand* validation rather than validation *en-masse* is the robustness of the approach with respect to ISR software evolution. If a new release of ISR client software uses a local software component that was not used in previous releases, Trust-Sniffer will discover its use even if a software developer fails to mention it in a checklist. If the integrity of any component cannot be validated, the user is alerted and the ISR resume sequence terminates.

2.2.2 Example Use

The following example illustrates the use of the system. Bob boots up a PC at his hotel's business center using his USB key. Trust-Sniffer's initial scan quickly validates the on-disk operating system, which is subsequently booted. Bob then initiates the resume step of ISR. Once Trust-Sniffer verifies the integrity of the ISR client software, Bob's personal execution environment is fetched over a trusted communication channel. The locally installed Web browser is riddled with malware. However, this does not affect Bob's task, because he works only within his trusted personal execution environment.

2.2.3 Threat Model and Assumptions

The ISR model facilitates the use of pervasive hardware deployments, but also must address the concern that physical access to unmanaged hardware is unrestricted. Machines used as ISR clients are vulnerable to attacks such as modifications to client or system software, or installation of malware such as key logging or screen capture software. Trust-Sniffer's goal is to avoid potential loss or disclosure of user data, by validating software on an ISR client machine before fetching the user's personal execution environment.

Trust-Sniffer's focus is on software attacks. While Trust-Sniffer is designed to safeguard mobile users, it can be coupled with security technologies used by system operators, such as full disk encryption, DMA monitoring and remote attestation. It does not prevent hardware attacks, which could be prevented by using tamper proof hardware or physical

surveillance. In addition, physical attacks such as "shoulder surfing" are outside of the scope of Trust-Sniffer's protection, and can be addressed using products such as screen protectors. Trust-Sniffer requires that the user is allowed to reboot the untrusted machine, and also that the BIOS allows booting from a USB memory stick. Since Trust-Sniffer was developed under the assumptions that modifications to the BIOS are difficult, it does not guard against virtual machine based attacks, such as SubVirt [30], where a compromised BIOS could boot directly into a malicious virtual machine monitor.

In the current version of the system, it is necessary for operating system kernels on untrusted machines to be equipped with appropriate software to carry out program validation. Details of the software required for program validation are outlined in Section 2.3. However, Section 2.4 describes how this assumption can be eliminated in future versions of the system.

Finally, since Trust-Sniffer is based on *load-time* binary validation, it does not protect against run-time attacks. Specifically, only applications with valid signatures are permitted to load and execute, but attacks such as buffer overflow attacks initiated after execution begins would not be detected.

2.3 Detailed Design and Implementation

Trust-Sniffer consists of three major components: (i) the trust initiator device and its associated minimal boot software; (ii) the trust extender, implemented as a kernel module; and (iii) the trust alerter, a user space notifier application. This is shown in Figure 2.2. This section describes the details of how the trust initiator is used to establish a root of trust on the untrusted machine, and subsequent extension of this root of trust by the trust extender. It also describes the details of the interaction between the trust alerter and the trust extender.

2.3.1 Integrity Measurement Architecture

Trust-Sniffer builds on the implementation of an Integrity Measurement Architecture (IMA) for Linux [47]. IMA is an instantiation of the trusted boot process, and uses the TPM to store system integrity measurements. IMA checks the integrity of the system software stack by computing a SHA-1 hash over the contents of an executable when it is loaded. This SHA-1 hash is referred to as a *measurement*.

IMA is built into the kernel, and is invoked as necessary when any executable is loaded,

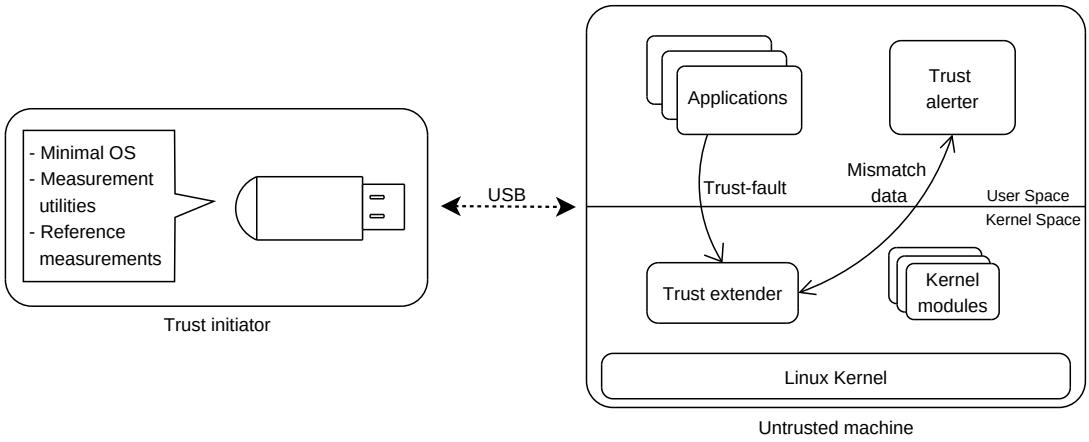

Figure 2.2: Trust-Sniffer architecture. The trust initiator plugs into an untrusted machine, validates its OS, and equips it with a list of SHA-1 hashes (reference measurements) of trusted applications. Applications that have not been validated cause a trust fault, which is handled by the trust extender. If an application cannot be validated, the user is notified via the user space trust alerter.

including user applications and kernel modules. Each loaded executable is measured and its hash value stored in a list inside the kernel. An aggregate measurement of the list is stored in the TPM. The TPM is used to protect the list's integrity by preventing malicious software on the machine from altering measurements. In addition, the TPM aggregate along with the kernel measurement list facilitates attestation of the system software stack by a remote party.

One important point to note is that IMA does not assume that the level of integrity provided using such a method guarantees that programs are completely invulnerable to attack. A program's static fingerprint at load time does not mean its behavior cannot be altered at run time. For example, programs are often modified at run-time using configuration files, which is another avenue for attackers to exploit program vulnerabilities.

Trust-Sniffer builds on IMA to suit the model of relegating security decisions to users. Since remote party attestation is not the intent of the system, it does not require special hardware such as the TPM. Instead, Trust-Sniffer uses a small, user carried passive device to initiate the process of establishing confidence in the bootloader and kernel on the untrusted machine. The OS kernel on the machine is then responsible for measuring software applications and preventing untrusted software from executing.

2.3.2 Validating Applications

Trust-Sniffer validates an application by comparing its *sample* measurement, obtained when the application is loaded, to a known list of *reference* measurements generated from trusted applications. An inequality between a sample measurement and every reference measurement in the list is called a *mismatch*. Note that in contrast to *attestation* where measurements are used to detect untrusted software after it has been loaded, *validation* refers to detection of untrusted software before it is used.

The reference measurement list has to be generated initially and updated when patches and new software are released. Note that failing to update the measurement list only implies that execution of new trusted software will be prevented. Execution of untrusted software is never allowed. For simplicity and compatibility, the measurement list format is the same as that output by `sha1sum`, a utility commonly available on most Linux distributions.

The initial prototype records measurements for the latest version of an application. However, it is conceivable for the list to contain multiple trusted measurements for an application such as the kernel. In this case, if a system with multiple kernels were encountered, the system could permit the execution of a trusted kernel even if the remaining ones were compromised. Note that since the majority of a user's execution environment is fetched by ISR client software, only measurements for a small set of OS and client software need to be maintained. To address the problem of frequent software updates and patches, users could obtain new measurements from a server that periodically generated updated lists and digitally signed them for distribution. This is discussed further in Section 2.5.

2.3.3 Rapidly Establishing a Root of Trust

Initially, none of the software on the target machine is trusted. IMA, as described above, needs additional software such as *TrouSerS* [9], an open source trusted bootloader and generic TCG based software stack, to validate the boot process. In order for measurements to be reliable, the system must ensure that the target machine's kernel is not compromised. Instead of relaying on a TPM, Trust-Sniffer makes use of the trust initiator's boot to validate the kernel and associated boot software. The goal of the trust initiator is to (i) bootstrap the trust establishment process by establishing a root of trust; and (ii) equip the on-disk OS with necessary tools to validate the rest of the software on the machine. These tasks are accomplished as rapidly as possible to avoid a long delay before the user can do useful work.

The trust initiator is partitioned, formatted and loaded with the Finnix bootable OS, a derivative of Knoppix. Partitioning and formatting USB memory sticks depend on the vendor and the drive geometry[1], and the details are omitted here. Finnix provides excellent support for devices, and automatic hardware detection. It is suitable for the Trust-Sniffer system because it boots quickly and has a small footprint (less than 100MB). It also has native support for Logical Volume Management (LVM) [55], which many Linux distributions use to manage storage. The trust initiator is loaded with the list of reference measurements for the initial boot validation phase as well as subsequent application validation by the on-disk kernel.

After the machine is booted using the trust initiator, all on-disk software components associated with the boot process are validated. A custom startup script mounts local hard disks and discovers the boot partition. It then uses `sha1sum` to measure the kernel image, the initial ramdisk, and the GRUB bootloader. The measurements of this set of software are compared to the list of reference measurements on the trust initiator. Any mismatches cause the boot process to be untrusted and halt the trust establishment process, since a trusted OS is required to validate the rest of the software stack. If all boot software is found to be valid, the reference measurement list on the trust initiator is copied to a predetermined location on disk, which is subsequently accessed by the kernel when it boots.

2.3.4 Dynamically Extending the Root of Trust

Once the root of trust has been established, the trust extender assumes the role of all validating application software, including dynamically loadable libraries, kernel modules and scripts. Use of the `kexec` [1] utility, which allows replacement of the current running kernel without using a bootloader, eliminates the need for the user to do a manual reboot (power off the machine, remove the trust initiator, and then power the machine on).

The trust extender kernel module uses the `securityfs` pseudo filesystem to communicate with user space programs. This is used to provide an interface for a user space program to load the trust extender with the list of reference measurements obtained from the trust initiator.

One important point to note is that the trust extender can begin enforcing measurement mismatches only after the reference measurement list has been loaded. It is thus imperative that this list be loaded as early in the boot process as possible. Until the list is loaded,

[1] Although the geometry of a USB memory stick is not physically comprised of cylinders, heads and sectors, the stick is encoded with logical CHS information. While older BIOSes used CHS as the mode of addressing drives, newer advanced BIOS implementations use logical block addressing (LBA).

User	**Trust Initiator**	**Trust Extender**	**Trust Alerter**
wants to use App_i	possesses *RM*		
inserts *TI* into untrusted ISR client			
	validate(*OS*)		
	boot *OS*		
	$\xrightarrow{RM}$		
		validate(*Init*)	
launches App_i			
		validate(App_i)	
		if *M*(App_i) $\in$ *trusted list*	
		run(App_i)	
		else	
		block(App_i)	
		$\xrightarrow{Mismatch}$	
			alert user

Figure 2.3: Information flow across the system. When the OS boots, failure to validate all measurements of programs launched during *Init* causes the kernel to panic. This is omitted for simplicity. RM = Reference Measurements

the trust extender is non intrusive, and allows all execution. A custom initrd image with modified boot scripts launches a user space utility to load the reference measurements into the trust extender. All initial execution before the list is loaded is from the initrd, which is part of the root of trust. Once the reference measurements are loaded, the trust extender protects the integrity of the in-kernel list by disabling the measurement loading interface in `securityfs`.

The reference measurement list is stored in a hash table indexed by the SHA-1 hash value of each executable. The trust extender uses the `file_mmap` Linux Security Module (LSM) hook to measure files mapped with the `PROT_EXEC` bit set (which includes dynamically loadable libraries and executables), and a custom hook in the `load_module` function to measure kernel modules. The LSM interface is part of the main kernel. Each measurement consists of the SHA-1 hash value of the executable code, and some additional file metadata such as pathname, user ID and group ID is also stored. The metadata does not affect the actual measurement validation procedure, but it is used to communicate information to the user. This is described in the next section. When a trust fault triggers the measurement of an executable, its value is compared against the reference list. If there is no matching reference measurement, the measurement is considered untrusted, and the kernel disallows the execution.

A caching mechanism is used to reduce the overhead of measuring unchanged applications in the case where they are executed more than once. Measured binaries are tagged for future reference. Each tagged value is stored in a special security substructure of the inode datastructure in the kernel, and is also part of each entry stored in the measurement list in the kernel. When a file that was previously measured is executed again, the kernel simply uses the cached inode value if it is not stale. A cached measurement becomes stale if a file is opened for writing, or if it is on an unmounted filesystem. Stale measurements need to be recomputed.

2.3.5 Alerting the User

When an untrusted application is encountered, the kernel blocks its execution. The user is not aware of communication between applications and the kernel, and needs to be alerted of relevant information in two instances: (i) when the trust extender encounters an untrusted application; and (ii) when a failure occurs in the kernel that would cause the measurement process to become compromised. The second case encompasses unexpected failures in the measurement process, such as out of memory errors.

To establish a channel for communication between the trust alerter and the trust exten-

der, Trust-Sniffer relies on the netlink socket interface. The netlink interface is a bidirectional, versatile method to pass data between kernel and user space, and the interface is well documented. The implementation uses a new netlink protocol type, `TS_NETLINK`, in addition to the `TS_NLMSG_MISMATCH` and `TS_NLMSG_FAILURE` message types. When the trust extender is initialized, it opens a netlink socket using a call to the function `netlink_kernel_create`. A user process that binds to this socket waits for messages from the kernel using the `recvmsg` system call.

When an untrusted application is encountered, the trust extender sends the appropriate message with a call to `netlink_broadcast`. In the case of a mismatch, this includes the process id of the process that failed, and the filename of the application on disk, if available. The current version of Trust-Sniffer includes a simple console application that communicates with the kernel. This can be converted into a more friendly graphical user interface application in future versions of the system.

2.4 Evaluation

This section evaluates Trust-Sniffer in light of user expectations about performance, and the security guarantees provided by the system. The discussions in this section serve to revisit the choices made in designing the system and assess their effectiveness in its overall usability, security and extensibility.

2.4.1 Security

In accordance with design goals presented in earlier sections, the system needed to strike an appropriate balance between ease of use and security guarantees. Currently, users have no choice but to trust any unmanaged hardware that they encounter. Trust-Sniffer provides a simple solution designed for use by non-experts, and security guarantees that are a substantial improvement over current practice.

Trust-Sniffer is not foolproof and is vulnerable to certain attacks. It does not guard against a malicious BIOS or virtual machine monitor, which could both provide the illusion that the machine was booting from the USB device even if this were not actually the case. It also assumes that the hardware is trustworthy and hence does not protect against DMA based attacks by malicious devices. Finally, some local configurations could cause increased security risks. For example, if a machine were configured such that several ports were left open and the firewall disabled, Trust-Sniffer would not be able to detect or

```
[asurie@shackleton]$ xterm
Killed
[asurie@shackleton]$
```

(a) Execution of an untrusted application is terminated

```
[asurie@shackleton]$ ./trust_alerter
Trust Alerter

The application xterm is untrusted and cannot
be validated.  Execution of process with Id
3535 has been blocked to minimize security risks.
```

(b) Notification to the user from the trust alerter

Figure 2.4: User experience when an untrusted application is encountered by the trust extender.

prevent network attacks.

Trust-Sniffer pessimistically disallows the execution of unknown software (i.e., software for which there is no reference measurement present). However, this should not inconvenience users. Establishing trust in a machine for use as an ISR client requires validation of only a small subset of software, since most applications are contained in the user's personalized execution environment that is downloaded from a trusted server. It is easy to equip the trust initiator with reference measurements required to validate software used by an ISR client.

Since the trust initiator is write protected, its software cannot be modified by unauthorized external sources. Of course, a user can disable the trust initator's write protection on his own machine to allow the reference measurement list to be updated. In addition, the trust establishment procedure does not depend on network communication using the untrusted machine; thus we disable networking capabilities on the minimal OS that we use to validate the on-disk kernel.

Figure 2.4 shows the output of Trust-Sniffer when it detects untrustworthy software. The kernel is loaded with a measurement list without the reference measurement for `xterm`, a commonly used application. Execution of `xterm` in shell produces the output shown.

Trust-Sniffer system should thus be useful in detecting and informing the user about

common occurrences, such as applications which have not been patched, or unrecognized applications which the user should not trust.

Measures the bandwidth attainable when reading from a cached file mapped into the processs address space

2.4.2 Performance

The Trust-Sniffer prototype is implemented using the Fedora Core 5 distribution, configured with version 2.6.15 of the Linux kernel. Since the system is implemented by extending a core part of the Linux kernel, the results discussed below should not be significantly affected by configuration changes, such as the use of a different Linux distribution. Test hardware for performance measurements consisted of an IBM T43 laptop with a 2.0GHz Pentium M processor, 2MB of L2 cache, and 1GB of RAM. A standard USB memory stick with 1GB of storage capacity was used as the trust initiator.

From the perspective of the transient usage model, it is necessary to evaluate how much overhead is required by the Trust-Sniffer system in comparison to a system without security checks. The metrics that matter most to the user are the time the system requires to boot, as well as the ongoing overhead required to check applications. The results below demonstrate that with Trust-Sniffer, the downtime caused by having to reboot is small and the additional security is worth the performance penalty. Note that the performance overhead for application measurements is only on the first execution.

Since the trust extender primarily uses the `file_mmap` LSM hook to measure applications at load time, the results include the latency associated with creating a mapping for a memory mapped file with the HBench-OS framework [20]. Table 2.1 shows the latency of a trust fault, with latency measurements averaged over 10 runs of the benchmark.

When an application is executed for the first time, its SHA-1 measurement is computed and this is denoted as a cache miss. If on subsequent executions an application's cached measurement is valid, its execution results in a cache hit. Baseline data indicates the latency of an `mmap` operation without Trust-Sniffer.

It is clear in the case of a cache hit, the overhead of validating an application that has previously been measured is not significant. When an application is first executed, there is some measurement overhead compared to the baseline. However, this overhead is low in absolute terms since applications are measured only once.

The average time it takes to boot a machine with Trust-Sniffer from the point the power button is pushed until a login prompt appears is shown in Table 2.2. This metric is relevant

Type	Latency (stdev)	Overhead
Cache hit	1.19 μs (0.07)	0.20 μs
Cache miss	4.29 μs (0.06)	3.3 μs
Baseline	0.99 μs (0.04)	–

Table 2.1: Latency of an `Mmap` Operation with the trust extender.

Configuration	Boot time (stdev)	Overhead
Trust-Sniffer	111.4 *sec* (0.52)	14.3 *sec* (14.2%)
Baseline	97.1 *sec* (0.57)	–

Table 2.2: Comparison of the time taken for the system to boot

to the user, since it defines the "warm-up" time needed by the system before the user can start doing work. Note that these results are somewhat dependent on system configuration, such as the number of daemons started when the OS is booted. However, they are sufficient to illustrate that the overhead of using Trust-Sniffer is small. The overhead of using Trust-Sniffer is minimal, only 14.2% over the baseline. One might argue that typically users are not required to reboot a system. However, the additional time spent to establish trust should be well worth the improvement in security.

Finally, as a point of discussion, there are a few ways to improve the performance of the kernel measurement mechanism. One observation is that there are likely to be performance gains if it is possible to reduce I/O overhead in reading a file to be measured. Currently, the measurement of an application is the SHA-1 hash of all of its executable code. If an application makes calls to any shared libraries, which is very often the case, the libraries also need to be loaded into memory and measured. The Linux kernel uses demand loading for executables, and when an application is executed, the required code is paged into memory on demand. For large applications, it may be the case that only a small portion of the executable code is mapped. This is true of library calls that the application makes, because typically an application does not require all the functionality of a given library. One way to reduce the I/O overhead of measuring applications could be to use a hash-tree based scheme to validate only the executable code of a file that is loaded. Under this scheme, the measurement of an application would be broken up into a series of individual hash values based on a predetermined block size, and measurements could be validated at a finer granularity. To accomplish this, the kernel measurement mechanism would have to be updated to hook into the page fault handler, so that only code

that was actually executed would be measured. Although this approach looks promising, its feasibility and security implications need to be investigated.

2.4.3 Usability and Extensibility

An important trend in computing systems is to facilitate use by both experts and non-experts. With Trust-Sniffer, the zone of trust expands from a small, convenient, trusted device that the user carries. Trust-Sniffer's operation model can help increase awareness and security for simple day to day computing tasks.

Trust-Sniffer's design is flexible and extensible. It should be easy to set up a mechanism for users to obtain updates to reference measurement lists. The prototype implementation uses Fedora Core; however, since the trust extender is implemented as a kernel module, the trust initiator could be loaded with appropriately configured kernel modules and initial ramdisks for stock kernels of various distributions. This would make it possible to equip arbitrary machines that did not have preconfigured kernels with Trust-Sniffer software.

Finally, although the discussion thus far has been in the context of Linux, the general approach is applicable to commercial operating systems, such as Windows. The SysInternals [7] project, a suite of system utilities for Windows, uses publicly available APIs which offer functionality similar to the APIs used by the Linux prototype described here. It should thus be feasible to implement a Windows version of Trust-Sniffer.

2.5 Discussion

With the trust initiator, a user essentially carries a minimal operating system on a USB memory stick. In this regard, one might consider it more useful to bring a full operating system and applications on the USB stick (or on CDROM/DVD, although these media types are mostly read only), which would eliminate the need for establishing trust in the untrusted machine. Other approaches to pervasive computing such as SoulPad [21] and Personal Server [61] provide a way for a user to carry his personal computing environment (including applications and data) with him on a mobile device. To avoid ambiguity, the discussion below refers to this as portable software.

There are a few reasons why this may not always be practical. First, portable software configurations would not contain correct local environment configurations such as printer and network settings that would be needed for correct operation. Modern operating sys-

tems are able to automatically detect certain settings, but this is not always perfect and the inability to access an essential service, such as the network, can be an enormous inconvenience to a user. Second, although automatic hardware configuration is improving and a large number of generic drivers are bundled with operating systems, it is not possible to guarantee that a portable OS will have every driver necessary for the hardware on an unknown system. Having a hash of each driver requires less space and is easier than ensuring that drivers are correctly configured for the hardware. This might prevent the user from being able to use common peripherals or devices. With Trust-Sniffer, having the necessary drivers on the trust initiator is not as big a concern. Establishing the initial root of trust requires only a basic set of hardware on the machine. It is assumed that as long as the trust initiator software is reasonably up-to-date, common devices such as disks should be well supported. Network and wireless cards are not required to be operational for the trust establishment process (and they are often the devices for which specific drivers are not bundled with operating systems).

Keeping the software on the trust initiator updated should not be difficult for users. Since the boot software is minimalistic, it should be easy to patch and update as new versions are released. Also, the list of trusted measurements has to be updated as vendors release patches or new versions of software. This includes the addition of new measurements, and the purging of old measurements that are no longer trusted. However, the list of reference measurements for commonly used applications is likely to be small.

One could imagine an infrastructure to facilitate the update process. The NIST-run National Software Reference Library project provides a database of 11,946,027 unique signatures (as of June 2007) of known benign and malicious software [5]. The database is updated quarterly and includes file signatures for several different operating systems and applications. Trusted entities could provide secure servers from which appropriately updated measurement lists could be obtained. For example, in a university or corporate campus deployment, administrators would be responsible for the maintenance of update servers. Users could then update the trust initiator by simply running an application that obtained the measurement list from the appropriate server.

Another point of discussion is that since the Trust-Sniffer system is not foolproof and does not offer perfect security guarantees, users may get a false sense of security. In particular, the security of the system relies on the trust initiator being updated regularly, which may make it vulnerable to zero-day attacks. The update problem is no different from the use of personal desktop operating systems today. If a user does not patch a security hole immediately (or the software vendor does not release a timely patch, which is not unusual) a zero-day attack is still possible. Compared to usage of untrusted systems today where users take no precautions at all, Trust-Sniffer substantially increases security

without placing a burden on the user. In addition, the process of using our system makes users conscious of risks they face and increases overall awareness about security.

2.6 Chapter Summary

Trust-Sniffer is a system to help users establish confidence in untrusted ISR clients. This chapter explores the security gains that can be achieved with Trust-Sniffer while placing as little burden as possible on the user. The system is built around a trusted user-carried passive device. The user device is small and inexpensive, and the system does not require any special hardware. Although some security guarantees are relaxed to achieve simplicity and ease of use, overall security for users is greater than it is today.

Chapter 3

Low Bandwidth VM Migration

ISR's tight encapsulation of user and execution state in a VM simplifies many aspects of moving computation from one site to another. However, when bandwidth is scarce (low-bandwidth, wide area networks) or expensive (cellular networks, such as EVDO), transferring such a large amount of VM state can be problematic. Such conditions are fairly common in the typical usage scenarios enabled by ISR, such as a user migrating his parcel from his home over a bandwidth constrained link to his office. This chapter explores *opportunistic replay*, a technique that optimizes the migration of VMs between frequently visited hosts. Opportunistic replay builds on the key observation that replay does not have to perfect to be useful.

Opportunistic replay involves first capturing a user's interactions with a VM, and then replaying these recorded interactions on a VM in the same initial state located at a remote site. The operating principle is that user actions are responsible for generating new state; hence, under ideal conditions, replaying a user's interactions on the remote VM should eliminate the need to ship state changes across the network.

Opportunistic replay is presented in the next several sections. The first section introduces necessary background concepts. The next section describes in detail how interactive replay is used in the context of VM migration. The third section discusses the challenges of this approach. The remaining sections describe an experimental implementation of opportunistic replay that addresses an initial subset of the challenges identified. The final section evaluates the performance of the system when migrating a VM over different domestic-class bandwidths and performing different types of representative end-user workloads.

3.1 Background

3.1.1 Content Addressable Storage

Content addressable storage (CAS) is a well known technique used to efficiently index and access common data blocks in large volumes of data. With CAS, cryptographic hashes of data blocks are computed, and blocks are named according to their hash value. CAS relies on the cryptographic hash property of *weak collision resistance*, and hence assumes data blocks with differing content will generate different hash values.

CAS has been previously used successfully to represent the common blocks of files in filesystems such as Venti [45] and LBFS [38]. It has also been used to improve caching of dynamic content over wide area networks. More recent work has explored its ability to improve the efficiency of large scale storage systems [19]. Previous work investigating the use of CAS in an ISR context demonstrated that it could provide significant savings in both storage requirements as well as network transfer times [40]. Although it has not been fully implemented in the current version of ISR (OpenISR 0.8.3 as of this writing) the testbed implementation described in this chapter relies on content addressability techniques to economize network utilization.

3.1.2 Interactive Log Based Record/Replay

Interactive log based replay is a technique where user interactions with a graphical user interface (GUI) application or environment are captured to a log, which can later be replayed to produce the same sequence of user actions. In this context, user interactions consist of key presses and mouse clicks, which represent the two most common modes of user input to application software.

Interactive record and replay techniques have commonly been used for tasks such as GUI test automation or automated software demonstration. There is a multitude of off-the-shelf software such that uses this technique for testing software written in a specific language such as Marathon (Java) [3], TestWorks (C/C++) [53] or software of a specific type, such as Badboy or Verisium vTest (web applications) [12, 14]. More generic functionality is provided by software such as GNU Xnee (Linux) [48], Eventcorder (Windows) [13], and VNCPlay [63] or VNCRedux [32] (platform independent). Such generic replay mechanisms are ideally suited for VM replay and state capture, since no specific access to the guest operating system or applications is required.

One of the more challenging tasks during replay is ensuring that when recorded actions

are replayed, the target application is in the correct state. For example, consider a user action that involves clicking on a menu button. If the menu has not yet appeared (for various reasons: the application may not have loaded yet, or the menu's appearance is dependent on a previous task/event that has not yet completed) replaying the action will be incorrect. Let us refer to this as the "missing menu" problem. To ensure correct replay, a variety of synchronization techniques are used. Langauge or special purpose replay tools use various specialized markers, and generic tools rely on screen state, or events from the windowing system for synchronization. However, despite using sychronization, non-deterministic events can affect the accuracy of interactive replay. This is described in more detail in the next section.

3.2 Challenges of Virtual Machine Replay

Many factors complicate the conceptual simplicity of opportunistic replay. These can be grouped into three broad categories that discussed below: (i) incomplete log capture; (ii) non-deterministic externalities; and (iii) exactly-once size effects.

Incomplete log capture: An obvious prerequisite for ideal replay is complete capture of all external stimuli that could perturb VM state. This includes external interrupts and data transfers from storage devices and networks, as well as user input via keyboard, mouse and other human interaction devices. A potential concern is the size of the log necessary for complete capture. In their work on VM logging and replay for intrusion analysis [23], Dunlap et al. report log growth ranging from 0.04GB per day to 1.2GB per day. Further, VMware report for their ReTrace tool a compressed log size of roughly 776 KB, without accounting for network activity, when rebooting a Windows XP VM [62]. Both sources report only modest CPU overhead for complete logging. However, logs growing at these rates would be impractical for VM migration, since they can easily surpass the size of the actual state changes that would be shipped with standard VM migration techniques. An additional concern pertains to inserting logging code in closed-source VMMs such as VMware Workstation, and closed-source guest operating systems such as Windows XP. Without access to the source code of these components, it may be impossible to ensure complete log capture. Capturing just user interaction is simpler, since the windowing system provides a natural interface for interposition of logging code. This will produce short logs that, even though incomplete, can still be leveraged by opportunistic replay to realize significant reductions in VM state transmission.

Non-deterministic externalities: Deterministic code execution is another obvious requirement for replay to produce VM state identical to the original execution. A major source of non-determinism in interactive systems is network access to Web sites with dynamically generated content. Consider, for example, Web access to a site that offers current stock quotes. Replaying the Web access may result in different content being returned because stock prices have changed. Even when the intrinsic content of a Web page is unchanged, there may be parasitic content such as advertising that is different on replay. Such non-determinism typically affects the memory component of VM state. However, this non-determinism may be propagated to the VM's disk state by user actions such as saving a screen image in a file. While there is no "solution" per se to the problem of non-deterministic externalities, opportunistic replay is able to cope because replay does not have to be perfect to be useful. Only when there is an excessive amount of non-determinism will the overlap between local and remote VM state drop below a useful level.

Exactly-once side effects: A difficult problem for VM replay is the occurrence of events that should not be replayed for reasons of correctness. Consider, for example, an interactive session in which a user sends an email message. Replaying this action would result in duplicate message transmission, which clearly violates correctness. A safe solution is to block any outbound network traffic during replay; this will result in divergent and less beneficial replay, but will guarantee consistency with the outside world. A solution with better replay performance would detect logged actions that have such exactly-once side effects and skip them during replay. Although this is a very difficult problem in its most general form, it may be relatively simple to perform conservative detection of common cases. For example, if there are a set of known Web sites at which a particular user performs financial transactions, the log can be examined for operations that reference these Web sites. An even more conservative approach is to suppress replay of all secure Web operations, on the grounds that high-value transactions are almost certain to occur only within the scope of a secure Web session. Finally, one could analyze network traffic and permit "read-only" transactions on well-known protocols, such as POP3's `STAT`, `LIST` and `RETR`, but not `DELE`.

3.3 Prototype Implementation

In the ensuing sections, *source* refers to the host at which a user is currently interacting, and *destination* is the host to which the VM will eventually be migrated. It is assumed

that both source and destination hosts have initially identical copies of a suspended VM. Opportunistic replay for VM migration consists of three phases: (i) capture; (ii) replay, and (iii) synchronization. These phases are not always sequential, depending on the replay strategy employed and other optimizations.

3.3.1 Operational Overview

- **Capture:** All keyboard and mouse input to the locally running VM is recorded to a log. The recording tool is configured to ignore user interactions sent to the host OS. While the interactive session is being recorded, the virtual disk keeps track of dirty disk blocks and the order in which they were modified. This makes the synchronization phase that follows more efficient.

- **Replay:** The recorded interaction log is transferred to the destination and replayed against the VM at the destination. The log may be replayed at the destination at a faster speed than the speed of capture at the source. This strategy exploits superior compute resources at the destination, and also suppresses think time during replay. As an additional performance enhancement, modified disk chunks at the source are shipped to the destination in the background while replay is in progress. This ensures that network bandwidth is not wasted by being idle during replay. Since replay may be imperfect, state produced at the end of replay may diverge more than state produced at the beginning. To optimize for this tendency, dirty disk chunks are shipped to the destination in reverse of the order that they were modified at the source. New VM disk state generated by replay at the destination is also tagged for synchronization.

- **Synchronization:** After replay, any residual memory and disk state differences at the source must be transferred to the destination so that the resulting state at both hosts is identical. Under ideal conditions, the amount of overlapping state generated at the source and destination is large, so there is little residual state to be transferred. Differences between the state at the source and destination are detected via cryptographic hashing using the SHA-1 algorithm, and propagated to the destination.

3.3.2 Replay Strategies

There are two different strategies for log based replay, based on the length and frequency of the capture and synchronization phases described earlier. These are depicted in Figure 3.1.

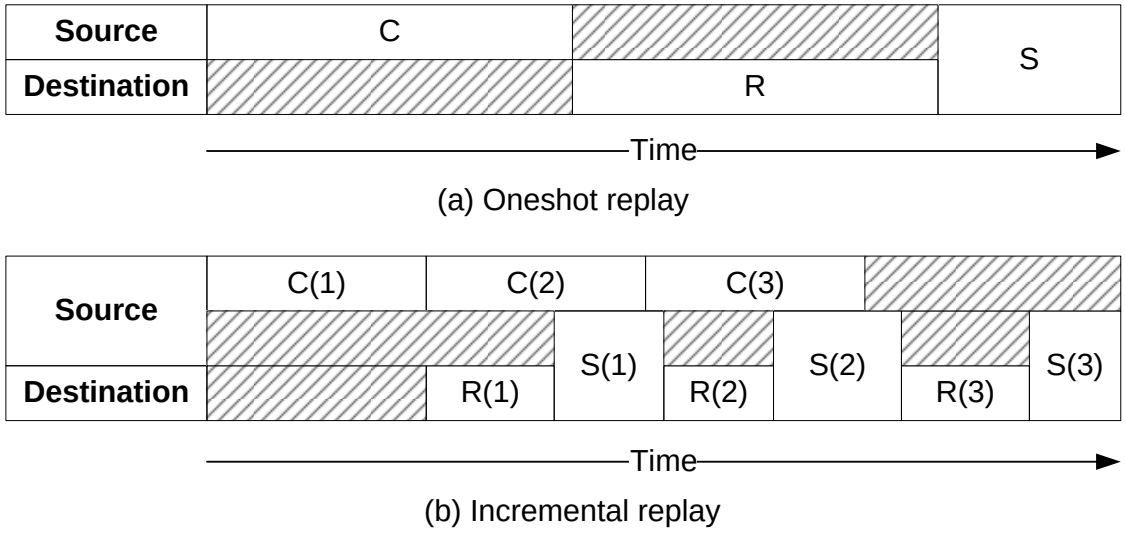

Figure 3.1: Different strategies for replay. C = Capture; S = Sync; R = Replay.

With *oneshot replay,* all user actions at the client are captured in a monolithic fashion into a single log. VM state at the destination becomes progressively stale as the capture phase proceeds. After the capture phase, the log is transferred to the server for replay and eventual synchronization. This process is depicted in Figure 3.1 (a).

Since replay is not always ideal, the likelihood that final VM state after replay at the destination diverges from state at the source increases with the length of the log. In addition to the non-deterministic externalities described earlier, synchronization failures of the "missing menu" category become more frequent for lengthy interaction logs. With oneshot replay, if a failure of this type occurs, replaying the rest of the log does not usually benefit the migration process.

To mitigate this problem, *incremental replay* divides a single user work session into a number of segments of shorter duration, with each segment having its own interaction log. A snapshot of VM state (memory and disk) is taken after each segment is captured at the source. Before replaying a given segment at the destination, VM state at the migration target is synchronized against the appropriate snapshot. This ensures that replay of each segment begins from a consistent state, and also limits the amount of divergence between state at the destination and source for a given interaction segment.

As soon as the first segment has been captured, a snapshot is taken at the source and the corresponding log is shipped to the destination where it is replayed. This is shown

Command	Description
isrreplay-createguest	Initialize VM at source and destination hosts
isrreplay-launch	Launch local interactive session
isrreplay-sync	Synchronize local modifications with destination
isrreplay-purge	Remove all traces of a guest from source and destination hosts
isrreplay-kill	Terminate all instances of a running VM
isrreplay-reset	Reset a VM to a consistent state

Table 3.1: ISRReplay System Interface

in Figure 3.1 (b). The numbers shown in the figure indicate the index of each segment. After replay, the VM at the destination is sychronized against the snapshotted state at the source. This procedure is followed for the remaining captured segments. The capture, replay and sychronization phases all take place more or less concurrently (after the first segment is captured). As a result, VM state at the destination lags only slightly behind that at the source. This bears resemblance to the concept of trickle reintegration in the Coda file system [37]. After replay and synchronization is complete for the final segment, state at the destination and the source are identical.

3.3.3 Implementation Details

This section describes the implementation of a prototype system that supports VM migration using opportunistic replay. Table 3.1 describes the interface to the system. The system allows a user to initialize a new VM from a pool of preconfigured VMs. The user can then work on the VM and migrate it to a designated destination when convenient using (i) oneshot replay, (ii) incremental replay, or (ii) no replay. If replay is not used, all modified state is shipped to the destination.

The prototype focuses on optimizing the synchronization of modified disk state between source and destination hosts, using a custom virtual disk that efficiently tracks modified disk state. It uses Xen [18] version 3.1 as the virtual machine manager. For record and replay operations, the system leverages the Xnee [48] tool. Xnee supports replaying interactive sessions on any X server, and is capable of replaying faster than capture speed.

Figure 3.2 shows the system architecture. The virtual disk is composed of a block device driver called Nexus, coupled with a user space daemon called Vulpes. While Nexus and Vulpes are both part of the current OpenISR implementation, Vulpes has been significantly modified from its original state for our purposes. Nexus chunks, compresses

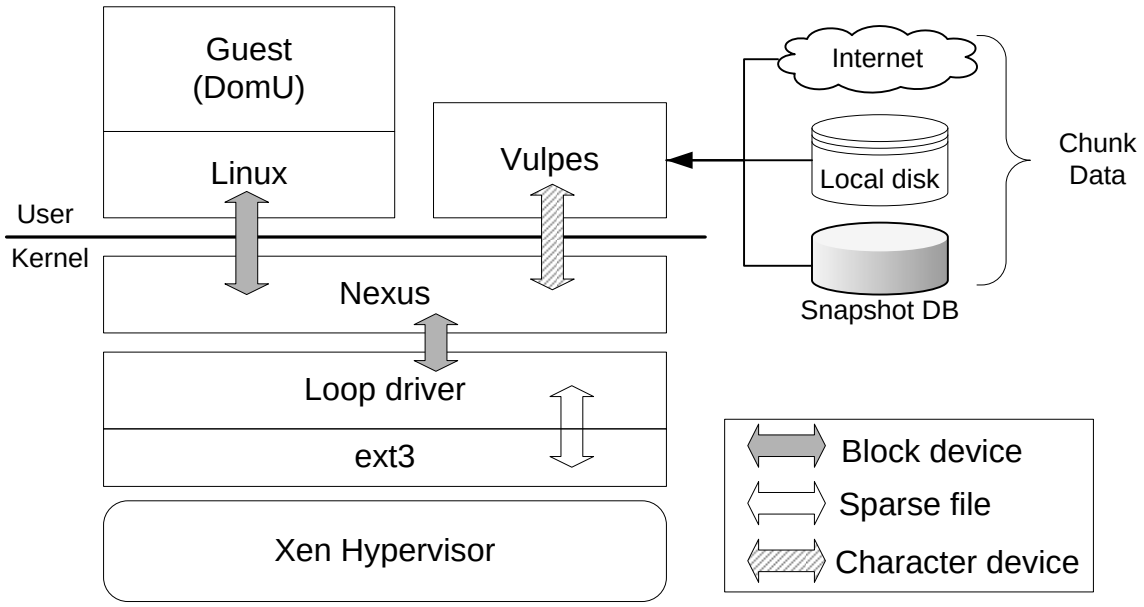

Figure 3.2: ISRReplay System Architecture

and convergently encrypts the data written to it. It uses Zlib [11] and Blowfish [51] as its compression and encrytion algorithms, respectively. Nexus exports a virtual disk to a VM as a physical block device, using the Linux loop driver with a sparse file as the backing store. Data is stored at a granularity of 4KB chunks. Previous work has shown that this maximizes the efficiency of both storage and network resources [40].

Vulpes uses a *keyring* to track and identify modified disk blocks. The keyring contains metadata associated with each chunk in the virtual disk, which consists of a *key* and a *tag*. The SHA-1 hash of a compressed chunk is called the *key*, which is used to encrypt the chunk. The hash of an encrypted chunk is known as its *tag*. Vulpes interacts with Nexus via a character device. Writes to data blocks are propagated directly by Nexus to the backing store, while Nexus informs Vulpes of updates to metadata. For reads, Nexus requests metadata from Vulpes. If the data does not exist locally, Vulpes fetches the data from the appropriate location, which may either be the local filesystem, a local chunk store, or the network. The backing store is then populated with the retrieved data.

Vulpes was extended to support virtual disk migration between a pair of hosts. Table 3.2 describes the list of commands recognized by Vulpes. Vulpes is run in client mode at the source, and in server mode at the destination. Communication between the client and

Command	Description
sync	Synchronize client state with server
send	Start background transfer of dirty disk chunks to the server
stop	Stop background chunk transfer
snapshot	Take a snapshot of the current disk state
connect	Attempt to reestablish server connection (only in client mode)

Table 3.2: Commands supported by the Vulpes disk daemon

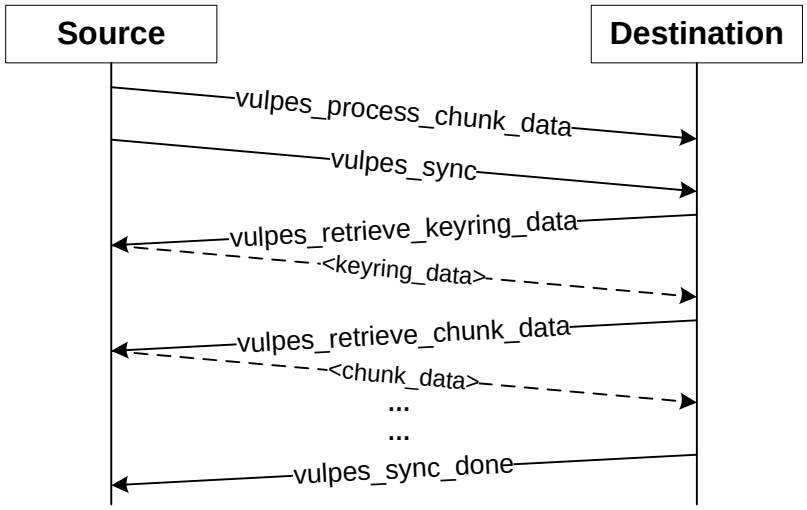

Figure 3.3: RPCs used for Virtual Disk Migration

Bandwidth	Upstream (Mbits/sec)	Downstream (Mbits/sec)
Cable	0.375	6
DSL	0.75	3
EVDO Rev. A	1.2	3.8

Table 3.3: Emulated Bandwidths

server is accomplished via an RPC package called MiniRPC, developed by the OpenISR team. Vulpes keeps track of the order in which disk chunks are modified. This enables disk blocks to be shipped in reverse order of modification, for the performance reasons described earlier. It supports taking snapshots of disk state, and keeps track of metadata associated with snapshots using a SQLite [6] database. The client is treated as having the authoritative copy of all disk state. Synchronizing two remotely separated virtual disks involves fetching all disk blocks not present locally at the server from the client. This process is depicted in Figure 3.3.

Dirty state tracking of virtual memory pages has not yet been implemented. This would couple the implementation with a specific VMM, and also requires source code access to the VMM. Instead, the system uses the `rsync` utility to efficiently synchronize saved memory images with oneshot replay. To ensure that `rsync` takes full advantage of similar data blocks, Xen's memory image format was modified to align saved memory pages on 4KB boundaries. To gather data on another alternative memory sychronizing mechanism, with incremental replay, the system uses the `xdelta` [10, 35] binary diff tool to transmit only the difference between memory images associated with successive segments.

3.4 Evaluation

I conducted several experiments using different workloads at various emulated bandwidths. In my experiments, the source host was configured with an Intel Pentium 4 3.60GHz CPU, 2MB cache and 2GB of RAM, and the destination host was configured with a 2.66GHz Intel Core 2 quad-core CPU, 4MB cache, and 4GB RAM. Ubuntu 7.04 was used as the guest OS and also as the host OS on both machines. The VM was configured with 512MB of RAM and a 4GB virtual disk.

Workload	Description
General	Web browsing Document editing
Install	Application installation
Kbuild	Kernel download & build
Gimp	Image manipulation

Table 3.4: Descriptions of Interactive Workloads

3.4.1 Experimental Setup

Table 3.4 describes the workloads used in the experiments. Tables 3.5 and 3.6 show the details of each of workload for oneshot and incremental replay, respectively. The General workload simulates a user conducting everyday tasks such as reading news websites (e.g, `CNN.com`) with Mozilla Firefox, and editing documents with the OpenOffice.org software suite and the Gedit text editor. For the Install workload, I install applications such as the Emacs text editor and the Mozilla Thunderbird email client using Synaptic, a graphical package managing tool available for the Ubuntu distribution. The Kbuild workload downloads the latest version of the Linux kernel from `www.kernel.org` and builds it without modules enabled. The Gimp program is used in the last workload to edit and manipulate sample images.

Each workload was acquired by recording a user's actions with a VM. As shown in the right hand columns of Tables 3.5 and 3.6, the size of the captured logs for each workload is three orders of magnitude smaller than the total amount of dirty state generated. For incremental replay, the length of each log segment is 120 seconds. Experiments were conducted using emulated bandwidths, shown in Table 3.3. The Cable and DSL bandwidth values were selected from the current offerings of two large ISPs. The EVDO Revision A standard is currently in use by wireless network providers such as Verizon and Sprint. Note that by emulating advertised peak bandwidths, the results bias against the benefits of opportunistic replay. In particular, since EVDO is sensitive to network congestion and proximity from cellular towers, bandwidths achieved in practice rarely approach peak bandwidths.

Each workload was run at the source host on VMs in the same initial state and the resulting VM state and interaction log were saved. For each experimental run, opportunistic replay of the saved log was performed at the destination, followed by transfer of residual disk and memory state from the source. For comparison, all modified memory and disk state was also transferred to the destination without replay. All results presented below are

Workload	Duration (minutes)	Log Size [Gzipped] (MB)	Dirty Disk State (MB)	Dirty Memory State (MB)
General	15.65	0.66 [0.15]	11.57	117.43
Install	2.85	0.11 [0.02]	57.08	105.18
Kbuild	7.23	0.20 [0.04]	153.51	144.47
Gimp	9.73	0.81 [0.18]	6.73	48.93

Table 3.5: Workloads for Oneshot Replay

Workload	Duration (minutes)	Log Segments	Total Log Size [Gzipped] (MB)	Dirty Disk State (MB)	Dirty Memory State (MB)
General	11.38	7	0.45 [0.10]	9.12	121.78
Install	2.85	3	0.21 [0.05]	57.08	114.85
Kbuild	7.23	4	0.37 [0.08]	198.45	146.08
Gimp	9.73	5	0.72 [0.16]	6.44	50.73

Table 3.6: Workloads for Incremental Replay

an average of 3 runs or more.

3.4.2 Results with Oneshot Replay

Replay at native speed: The first set of experiments demonstrates the benefits of replay when the log is replayed at native (capture) speed at the destination. Figure 3.4 and Figure 3.5 show the amount of disk and memory state transferred during migration with and without replay at the destination at various bandwidths. The figure shows that the wins are greatest for the Kbuild workload, where disk and memory state transferred at Cable speed shrink from 153.51MB to 29.74MB, and from 144.47MB to 28.9MB, respectively. As described in Section 3.3, dirty disk state is shipped in the background while replay is in progress at the destination. A higher upstream bandwidth results in more disk state shipped in the background and not being generated through replay. For example, at EVDO speeds, disk state transfer for the Install workload is reduced from 57.08MB to 44.73MB. In contrast, at cable speeds, disk state transfer is reduced to 14.06MB. If the amount of modified state is small or bandwidth is high, the background disk state transfer might finish before replay is complete, and replay is terminated; this holds true for the General and Gimp workloads. Early replay termination also limits the amount of memory state being generated. For example, with the General workload, 107.55MB of memory state is shipped at EVDO speeds, which is reduced to 58.63MB at Cable speeds. Thus, even for

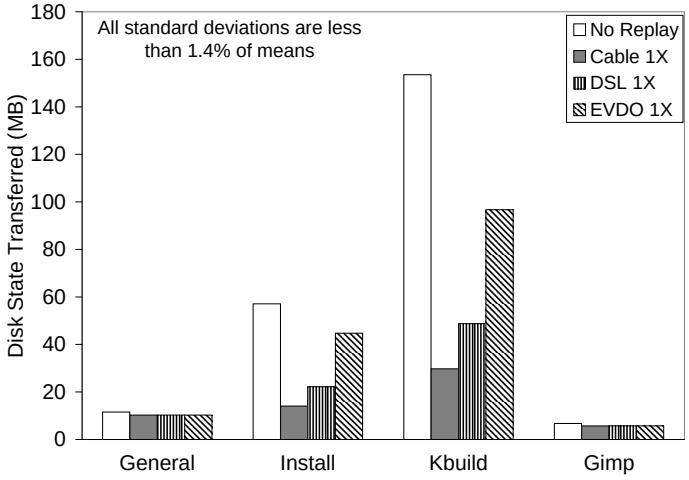

Figure 3.4: Disk bytes transferred with Oneshot Replay at native speed

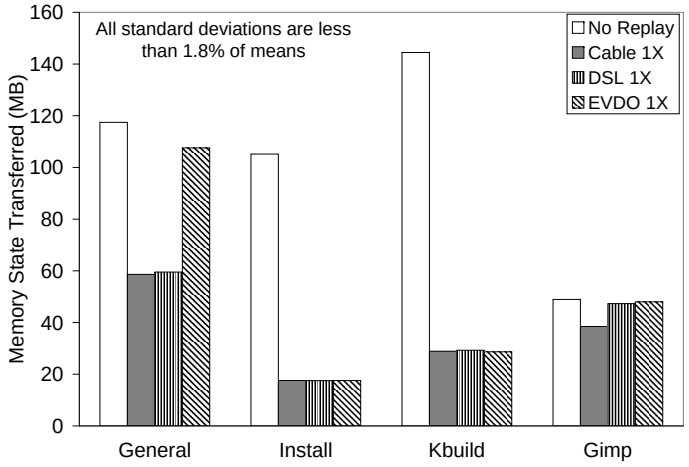

Figure 3.5: Memory bytes transferred with Oneshot Replay at native speed

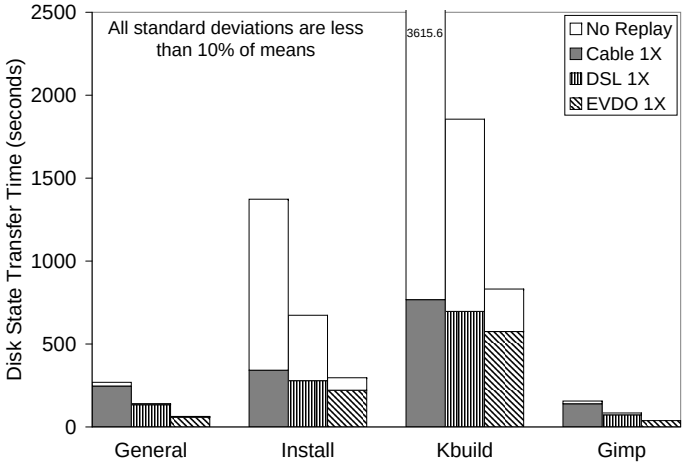

Figure 3.6: Total disk state transfer time with Oneshot Replay at native speed ("Replay" bars superimposed on top of "No Replay" bars).

interactive workloads that generate little dirty disk state, significant savings in terms of memory state shipping can be realized.

Figures 3.6 and 3.7 show the corresponding reduction in migration time using replay at native speed for both disk and memory state. Replay accounts for significant time savings in disk state transfer, which is most apparent at the lowest bandwidths. At Cable speed, transfer time for disk state reduces from 1373 seconds to 342 seconds for the Install workload, and from 3615.6 seconds to 767 seconds, for Kbuild. This trend is also true of memory state transfer, with the Install workload obtaining the largest reduction in transfer times at Cable speed. For Install, transfer times are reduced from 2459.8 seconds to 400 seconds. For the General & Gimp workloads, transfer times at Cable speed go down from 2744.5 to 1365.5 seconds, and from 1137.2 to 894.5 seconds, respectively.

Replay at speeds higher than native: The goal of higher speed replay is to suppress idle or think time during replay, in order to generate more VM state in less time at the destination. Figures 3.8 and 3.9 show the benefits of higher speed replay for General and Gimp, the most interactive workloads in my experiments. In the figures, 2X replay indicates that think time between interactions was compressed by half during replay. Only results for EVDO speeds are shown; by compressing think time, more of the log is replayed before the background transfer of disk state finishes. This reduces the amount of memory state transferred. The results at other bandwidths do not show improvements as the entire log is replayed even at native speed. Kbuild and Install also show little improvement, since

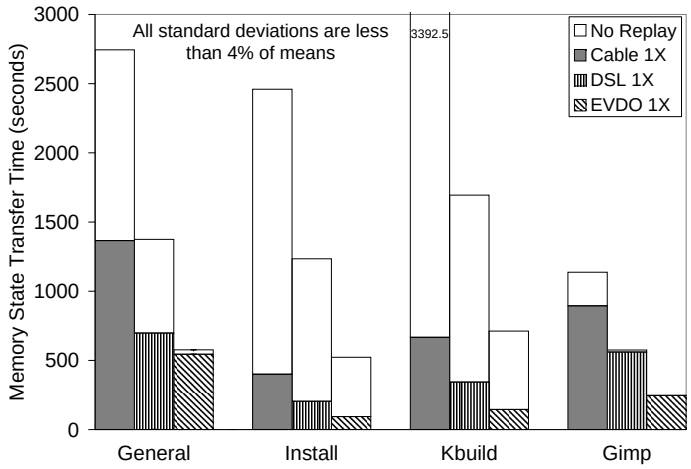

Figure 3.7: Total memory state transfer time with Oneshot Replay at native speed ("Replay" bars superimposed on top of "No Replay" bars).

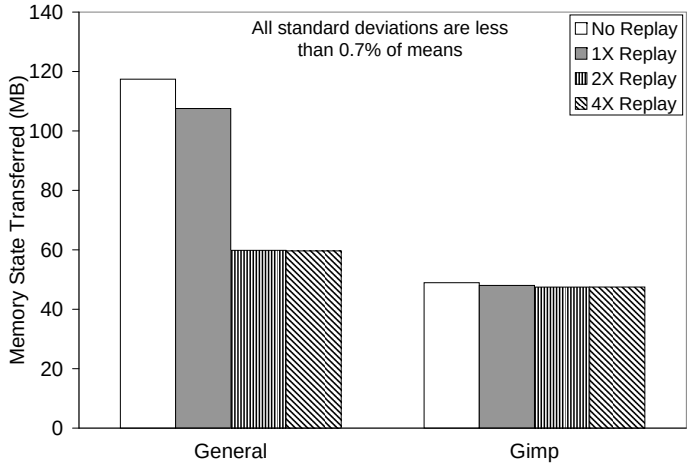

Figure 3.8: Memory bytes transferred at EVDO bandwidth with Oneshot Replay at higher speeds

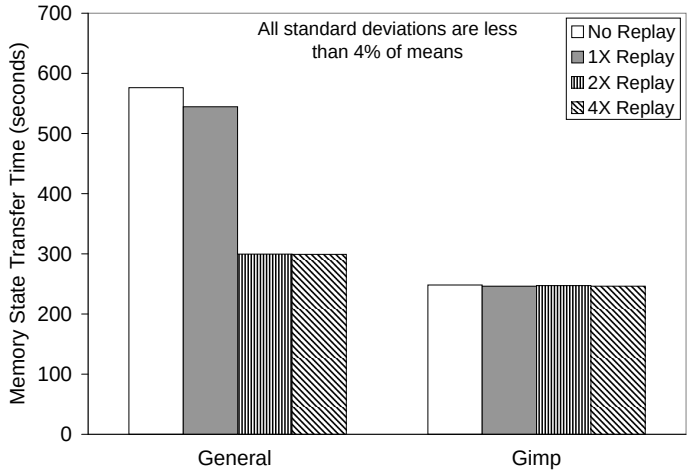

Figure 3.9: Transfer time for memory state at EVDO bandwidth with Oneshot Replay at higher speeds

Workload	No Replay (MB)	Replay (MB)
General	121.78	127.42
Install	114.85	111.68
Kbuild	146.08	230.94
Gimp	50.73	55.96

Table 3.7: Memory State Transferred with Incremental Replay

they are CPU intensive workloads that expose almost no compressible think times. Finally, the amount of dirty disk state for Gimp and General is small and not significantly affected by faster replay.

The General workload benefits from higher speed replay, with memory state transfer decreasing from 117.43MB with no replay to 59.64MB when the session is replayed at four times the native speed. The same trends translate to Figure 3.9, with transfer time decreasing from 576.17 seconds to 299.17 seconds. The Gimp workload does not significantly benefit from higher speed replay as it does not generate much dirty memory state.

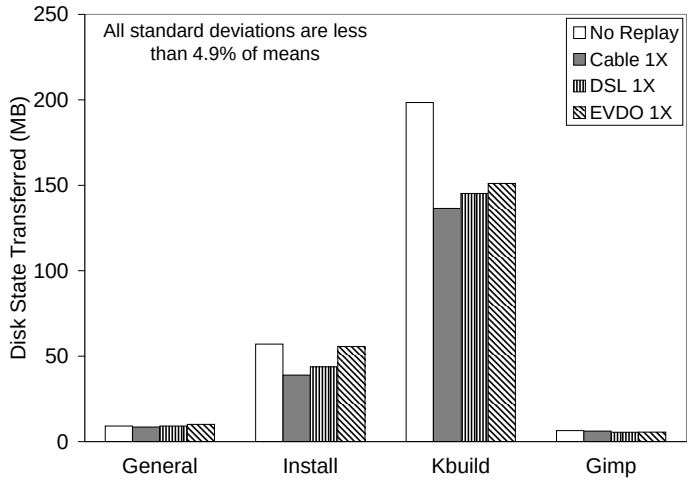

Figure 3.10: Disk bytes transferred with Incremental Replay at native speed

3.4.3 Results with Incremental Replay

Figure 3.10 shows the amount of disk state transferred at native speed using incremental replay at various bandwidths. As was the case with oneshot replay, the amount of state transferred goes up as bandwith increases, since replay is able to progress further before it terminated when background disk state transfer is completes. The greatest reduction in disk state transferred is realized at Cable speeds for the Install and Kbuild workloads. For Install, disk state transferred goes down from 57.08MB to 38.95MB, and for Kbuild, transferred state is reduced from 198.45MB to 136.41MB. There are no appreciable savings for the remaining workloads, since they do not generate much dirty disk state. Overall savings are less in comparison to oneshot replay, since more state is transferred due to sychronization after each segment is replayed.

One significant drawback of incremental replay is the amount of memory state that needs to be transferred due to per-segment synchronization of state. As shown in Table 3.7, with incremental replay, there is no reduction in memory state transferred, and in certain cases, more memory state is transferred in comparison to the baseline case when replay is not used. Overall state transfer times with incremental replay are worse than those without replay, since memory state transfer comprises a significant fraction of the total transfer time. Experimental results at higher replay speeds did not show much improvement, if at all, in comparison to native speed replay and are hence omitted here. Since the prototype system does not track dirty memory pages, it is possible that substantial implementational improvements can be made. However, the results presented here do not provide enough

concrete evidence in support of incremental replay.

3.5 Chapter Summary

This chapter presented opportunistic replay, a technique that replays user actions on a target site to recreate VM state and minimize the overhead of VM migration. The chapter presents oneshot replay and incremental replay as separate strategies for VM migration. While results for incremental replay indicate that further experimentation is necessary, the evaluation of oneshot replay demonstrates that even imperfect replay has the potential to achieve savings as high as 80.5% of bytes transferred and 80.6% of time saved, for workloads that generate large amounts of modified state. Even for interactive workloads that do not generate much disk state, replay reduces memory state transferred.

Chapter 4

Related Work

The work described in this document is unique in its focus on improving mobile infrastructure for pervasive computing. Although some of the techniques described here are similar to those used in other systems, to the best of our knowledge, they have not previously been used in the context of improving transient use of PCs.

This chapter is divided into two sections. The first section describes work in security related to Trust-Sniffer. The next section reports on work related to the mechanisms used to support low-bandwidth VM migration.

4.1 Secure Systems

When comparing the work described in this document to other work, note the following: (i) most systems are designed with an administrator in mind and are not focused on providing security guarantees to users; and (ii) although many solutions offer more security than Trust-Sniffer, they have complex hardware and software requirements, which slows their adoption in practice.

4.1.1 Boot Process Modifications to Enhance Security

The *trusted boot* and *secure boot* [17] mechanisms verify software components required during a PC's bootstrap process. However, both mechanisms require platform modifications, and do not verify software such as operating system services and user applications, which are executed after the bootstrap process is complete. In addition, although trusted

boot facilitates the detection of untrusted software through recorded application signatures, it does not prevent untrusted execution since signatures are not validated when they are recorded. Trusted boot is designed to allow remote attestation of system sofware by a third party. Trust-Sniffer builds on trusted boot to provide security to users, by implementing software validation at run time.

4.1.2 Systems Designed to Provide Security to Users

A well studied problem is the untrusted terminal problem. This is the problem of a user interacting with an untrusted computing device. There are two formulations of the problem. The untrusted device can be used as (i) a surrogate to establish secure communication with a trusted remote device, or (ii) a local kiosk which the user accesses to perform work, relying on locally installed software and peripherals. Trust-Sniffer represents a middle ground between these two categories. While it supports local software validation, it is intended for use with ISR, which fetches the user's computing environment from trusted remote servers.

Solutions that address the first problem include a camera-based authentication mechanism to establish an authenticated bidirectional channel between the remote device and a user on the untrusted terminal [22]. With visual cryptography, authenticity and secrecy of messages displayed on the terminal can be maintained without using additional hardware [39]. A number of solutions do not rely on the use of the untrusted terminal hardware, and instead use trusted hardware carried by the user, such as smart cards and PDAs [15, 26, 42, 44].

The Pioneer system addresses the second version of the problem using a challenge response protocol initiated by a *dispatcher* to provide verifiable code execution on an untrusted platform [52]. The dispatcher detects malicious tampering by measuring the response time of a verification function that runs on the untrusted platform. Pioneer allows allows for an attacker to have complete control over the software on a system, including administrative privileges, but makes several assumptions, notably that the dispatcher knows the hardware configuration of the untrusted platform in advance. Since Pioneer establishes a dynamic root of trust, it can be used as a building block for future iterations of Trust-Sniffer.

Garris et al. also address the untrusted kisok problem [25]. However, their implementation relies on secure hardware, including a TPM chip, and a new instruction recently added to the x86 architecture. In addition, their implementation requires validating all the software on a kiosk before it can be used, whereas Trust-Sniffer allows incremental

program validation to facilitate transient use.

Another strategy is to secure sensitive user input from malicious software, rather than validating all software on the kiosk [36]. With BitE, the user carries a trusted active device (a cellphone), that communicates with the unknown system to attest its software. Users enter sensitive input using the trusted device, which is communicated to individual applications over a secure channel. BitE differs from Trust-Sniffer in that is not intended to validate applications, and requires a TPM chip.

4.1.3 Systems Designed to Provide Security to Administrators

There is much work in the literature on systems designed to enhance security from an administrator's perspective. The Tripwire project provides software that monitors and audits changes to specific files from a known baseline configuration [29]. It detects changes to files by analyzing periodic snapshots of the file system. This solution is beneficial to system administrators, who often need to maintain standard system configurations for their users, and is not intended to provide end users with any guarantees on their personal data. In addition, its security guarantees are limited, because changes are not detected in real time. This means that an attacker could go undetected by compromising the system during an interval when the change detection agent is not running.

Kennel and Jamieson describe methods to remotely establish whether the hardware and software of a computer system are *genuine*, without relying on any additional hardware [28]. Like Pioneer, a remote system poses a set of challenges to the client, which the client should only be able to solve satisfactorily if its hardware and software match what the remote system expects. If the client successfully passes the challenge, it can safely be granted access to distributed network resources, such as an NFS server. This is in contrast to Trust-Sniffer, which is designed to protect users from malcious software, rather than infected machines from contaminating network resources.

Terra is an architecture for trusted computing based on virtual machines [24]. It aims to allow applications with different security requirements to run side by side on the same system in different virtual machines. The core of the platform is based on a trusted virtual machine monitor, which is responsible for allocating hardware resources among virtual machines. Trust-Sniffer, coupled with ISR, protects the user's computing environment from other malicious software by limiting a user's interactions to validated guest software.

Copilot is a coprocessor based DMA monitoring system that detects modifications to a host operating system's kernel [41]. Although such technology relies on special hardware and is not designed for mobile use, it is complementary to Trust-Sniffer. While Copilot

could be leveraged to provide security for system operators, Trust-Sniffer would safeguard mobile users.

4.2 VM Migration

4.2.1 Operation Based Update Propagation

Closest in spirit to opportunistic replay is the work of Lee et al. on operation shipping for mobile file systems [33, 34]. That work showed how logging and replay could significantly improve performance in propagating large files from a weakly-connected client to a server. Bayou's anti-entropy protocol for reconciling state between weakly consistent replicated storage systems also relies on propagating updates rather than full database contents between replicas [43]. The work described here differs in two major ways. First, the focus is on re-creating VM state rather than file system or database state. Second, this work uses an opportunistic approach and can therefore benefit even from replays that diverge from the original execution.

4.2.2 Data Similarity

The use of an opportunistic approach to exploiting data similarity was inspired by the work of Tolia et al. in distributed file systems [56, 57] and relational databases [58, 59]. A recurring theme in that work is the description of a large data object in recipe form using cryptographic hashes, and the synthesis of parts of that object from local data sources in order to reduce transmissions over a bandwidth-challenged network. More recently, Annapureddy et al. have used similar techniques in the context of cooperative caching for file servers [16]. The work described here extends the opportunism underlying these approaches to the realm of VMs, using techniques specific to log capture and replay. At an even deeper level, the idea of using cryptographic hashing for detecting similarity of data content has been used in file systems such as Venti [45] and LBFS [38], and the `rsync` file transfer protocol [60].

4.2.3 VM Replay

The concept of VM logging and replay was introduced by Dunlap et al. [23] in the context of the ReVirt system for intrusion analysis. Xu et al. also proposed deterministic VM re-

play for the purposes of acquiring execution traces for computer architecture research [62]. Since both systems are dependent on complete log capture and ideal replay, they require complex source code modifications to the VMM, and generate very large logs. In contrast, opportunistic replay can afford to be less strict. As the results from Chapter 3 demonstrate, a profitable level of fidelity of log capture and replay can be implemented without VMM modifications.

Chapter 5

Conclusion

Plummeting hardware costs, coupled with the emergence of pervasive computing systems has enabled ubiquitous access to personalized computing environments. In order to fully realize the vision of seamless mobility and remove the tethering between hardware and software, it will be necessary to efficiently and accurately "sanitize" a machine before it is accessed by the next user. In addition, despite improvements to network technologies in recent years, such as home broadband connections being replaced by fiber optic lines, systems will still continue to encounter conditions of poor connectivity.

This document has described two mechanisms to improve mobile infrastructure for transient use of PCs, a new usage model enabled by Internet Suspend/Resume. It has introduced Trust-Sniffer, a system to help users establish confidence in unmanaged hardware. Efficient migration of VM state is enabled via opportunistic replay, a technique that replays user actions on a target site to recreate VM state. Both techniques were prototyped in the context of Internet Suspend/Resume system, which is still being actively developed and enhanced with new features. Experimental data from the system verifies that these techniques can help users cope with varying conditions of security and connectivity encountered in mobile environments.

5.1 Contributions

The main contribution of this work is demonstrating how user influences can be leveraged to improve systems designed for mobile computing. The Trust-Sniffer system adopts an optimistic approach which relaxes some security guarantees but focuses on use by non-experts. Trust-Sniffer's implementation is simple and user friendly, and is based on a

small, passive device that does not depend on special tamper proof hardware. It provides a mechanism to incrementally validate user relevant portions of the software stack on demand, which facilitates transient use, as well as an efficient system to protect users from the inadvertent execution of malicious code. Finally, by providing the user with a simple metaphor for a complex problem, it helps increase user awareness about security.

This document also presents a novel application of interactive replay for VM migration. By acknowledging that replay for VM migration does not need to be perfect to be useful, opportunistic replay addresses the problem at the GUI level, greatly reducing complexity and producing short logs, while still yielding byte transfer reductions of up to 80.5%. Promising results reported with the early prototype suggest that it should be useful to include opportunistic replay in the production version of ISR and analyze its performance in a live deployment.

Finally, this work also contributes an implementation of the techniques described here in two experimental systems. Both systems can be used to refine the techniques described here for practical use. They can also be used as building blocks to experiment with new techniques.

5.2 Future Work

While this work presents encouraging results, there a number of research directions that remain unexplored. The rest of this section discusses directions for future work in both prototype systems developed during the course of this work.

5.2.1 Rapid Trust Establishment

Leveraging Processor Based Devices for Trust Decisions

The current version of Trust-Sniffer uses a passive device to evaluate the untrusted machine. While this allows the solution to remain inexpensive and facilitates ease-of-use, it limits security guarantees. In particular, with a passive device, only a *static root of trust* can be established. Using processor based devices, it should be possible to improve security by establishing a *dynamic root of trust*, with the aid of systems such as Pioneer [52]. This will eliminate Trust-Sniffer's current assumption that the BIOS is trustworthy.

Establishing Trust by Consensus Methods

Another potential area of exploration is using consensus decisions from multiple users to determine whether a machine is trustworthy. With ISR, a future expectation is that users should be able to access their personalized environments from anywhere, on any hardware. Suppose the same laptop at a coffee shop was used successively by different ISR users. If every user carried a version of Trust-Sniffer and communicated any local trust decisions to a remote server for access by other ISR users, collective decisions regarding the status of the untrusted machine could be made. Such a mechanism could be configured in a such a way that only users belonging to the same ISR deployment would be able to collaborate on trust decisions. Moreover, such consensus decisions would be used as an aid rather than the sole security mechanism employed.

5.2.2 VM Migration

Implementation Aspects

Tracking VM memory state as it is dirtied will further optimize the network transfer savings realized. In addition, the virtual disk used in the current prototype does not support disk state snapshots at run-time. It should thus be possible to improve performance by replacing the current implementation with a true copy-on-write virtual disk.

Eliminating Exactly-Once Side-Effects

Policies to prevent replay from altering precious outside-world state (such as email inboxes or bank accounts) need to be developed. There are are a few different ways that this could be accomplished. With the current system, logs could be analyzed for "dangerous" state modifying actions and the system could prevent these actions from being replayed. If related actions could be grouped together in a logical fashion, the system could prompt the user for advice when there was any doubt about the side-effects of a set of actions. Another strategy could be to hook into at the VMM level and conservatively disallow network activity that modified external state.

Application-Aware Replay

In the current system, replay is blindly opportunistic. While simply recording keyboard and mouse interactions produces short logs, the lack of additional information makes the

logging mechanism completely application unaware. The log does not preserve causal relationship between user actions and generated state, and as a result the replay mechanism is subject to synchronization failures. In many cases, this limits the benefits of replay. Future versions of the system could reduce the occurrence of this problem by increasing the fidelity of the log in a limited fashion. The log could be annotated with additional metadata such as the focus application window and files accessed by a sequence of user actions by interfacing with additional components in the system such as the window manager, file system and VMM.

5.3 Final Thoughts

Internet Suspend/Resume introduces the novel model of transient PC use. While there have been significant advances in mobile computing in recent years, enabling pervasive computing is still a work in progress. The goal of this thesis was to enhance a user's ability to make trust decisions in his local computing environment, and improve the performance of migrating a user's parcel between resume sites. This work shows that it is possible to implement these improvements to mobile computing infrastructure using load-time software validation and opportunistic replay as the underlying techniques. These techniques have been implemented in two experimental systems without source code modifications to operating systems or virtual machine monitors. The experimental results obtained through this work demonstrate that load-time software validation and opportunistic replay are promising techniques for real deployments.

Bibliography

[1] Reboot Linux faster using kexec. http://www-128.ibm.com/developerworks/linux/library/l-kexec.html. 2.3.4

[2] Knoppix. http://www.knoppix.net/. 2.1.1

[3] Marathon Java GUI testing tool. http://www.marathontesting.com. 3.1.2

[4] MiGO. http://www.migosoftware.com. 1

[5] National Software Reference Library. http://www.nsrl.nist.gov. 2.5

[6] SQLite. http://http://www.sqlite.org/. 3.3.3

[7] Windows sysinternals. http://www.microsoft.com/technet/sysinternals/default.mspx. 2.4.3

[8] Trusted computing group. http://www.trustedcomputinggroup.com. 2.1.2

[9] TrouSerS. http://trousers.sourceforge.net/. 2.3.3

[10] Xdelta binary diff. http://www.xdelta.org. 3.3.3

[11] Zlib compression. http://www.zlib.net. 3.3.3

[12] Badboy. http://www.badboy.com.au, 2004 . 3.1.2

[13] Eventcorder suite. http://www.eventcorder.com, 2007. 3.1.2

[14] vTest. http://www.verisium.com, 2006. 3.1.2

[15] Martín Abadi, Michael Burrows, Charles Kaufman, and Butler Lampson. Authentication and Delegation with Smart-cards. *Science of Computer Programming*, 21(2): 91–113, October 1993. 4.1.2

[16] Siddhartha Annapureddy, Michael J. Freedman, and David Mazires. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005. 4.2.2

[17] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 65–71, May 1997. 2.1.2, 4.1.1

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of the* 17^{th} *Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003. 3.3.3

[19] Deepak R. Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. Improving duplicate elimination in storage systems. *Trans. Storage*, 2(4):424–448, 2006. ISSN 1553-3077. doi: http://doi.acm.org/10.1145/1210596.1210599. 3.1.1

[20] Aaron B. Brown and Margo I. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 214–224, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-909-2. doi: http://doi.acm.org/10.1145/258612.258690. 2.4.2

[21] Ramón Cáceres, Casey Carter, Chandra Narayanaswami, and Mandayam Raghunath. Reincarnating PCs with Portable SoulPads. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, pages 65–78, New York, NY, USA, 2005. ACM Press. ISBN 1-931971-31-5. doi: http://doi.acm.org/10.1145/1067170.1067179. 2.5

[22] Dwaine Clarke, Blaise Gassend, Thomas Kotwal, Matt Burnside, Marten van Dijk, Srinivas Devadas, and Ronald Rivest. The Untrusted Computer Problem and Camera-Based Authentication. In *International Conference on Pervasive Computing*, 2002. 4.1.2

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002. 3.2, 4.2.3

[24] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of*

the Nineteenth ACM Symposium on Operating Systems Principles, pages 193–206, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945464. 4.1.3

[25] S. Garris, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang. Towards Trustworthy Kiosk Computing. In *Proceedings of 8th IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, February 2007. 4.1.2

[26] Howard Gobioff, Sean Smith, J.D. Tygar, and Bennet Yee. Smart Cards in Hostile Environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, Oakland, California, November 1996. USENIX. 4.1.2

[27] Keith J. Jones. Loadable Kernel Modules. *;login: The Magazine of Usenix & Sage*, 26(7):43–49, November 2001. 2.1.1

[28] R. Kennell and L. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of 12th USENIX Security Symposium*, pages 295–310, 2003. 4.1.3

[29] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In Jacques Stern, editor, *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, COAST, Purdue, November 1994. ACM Press. 4.1.3

[30] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. doi: http://dx.doi.org/10.1109/SP.2006.38. 2.2.3

[31] Michael Kozuch and Mahadev Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, June 2002. 1

[32] H. Andrs Lagar-Cavilla, Niraj Tolia, Eyal de Lara, M. Satyanarayanan, and David O'Hallaron. Interactive Resource-Intensive Applications Made Easy. In *Proceedings of Middleware 2007: ACM/IFIP/USENIX 8th International Middleware Conference*, Newport Beach, CA, November 2007. 3.1.2

[33] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based Update Propagation in a Mobile File System. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999. 4.2.1

[34] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation Shipping for Mobile File Systems. *IEEE Transactions on Computers*, 51(12), 2002. 4.2.1

[35] Joshua P. MacDonald. File System Support for Delta Compression. Master's thesis, University of California at Berkeley, 2000. 3.3.3

[36] J. M. McCune, A. Perrig, and M. Reiter. Bump in the Ether: A Framework for Securing Sensitive User Input. In *Proceedings of USENIX Annual Technical Conference*, June 2006. 4.1.2

[37] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *SOSP '95: Proceedings of the Fifteenth ACM symposium on Operating Systems Principles*, pages 143–155, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224068. 3.3.2

[38] A. Muthitacharoen, Brad Chen, and David Mazieres. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001. 3.1.1, 4.2.2

[39] Moni Naor and Benny Pinkas. Visual Authentication and Identification. In *Proceedings of Crypto97*, pages 323–336. 4.1.2

[40] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In *USENIX-ATC'06: Proceedings of the Annual Technical Conference on USENIX'06 Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association. 3.1.1, 3.3.3

[41] Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association. 4.1.3

[42] Alina Oprea, Dirk Balfanz, Glenn Durfee, and D. K. Smetters. Securing a Remote Terminal Application with a Mobile Trusted Device. In *20th Annual Computer Security Applications Conference (ACSAC'04)*, 2004. 4.1.2

[43] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*

'97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, pages 288–301, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. doi: http://doi.acm.org/10.1145/268998.266711. 4.2.1

[44] Andreas Pfitzmann, Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Trusting Mobile User Devices and Security Modules. *IEEE Computer*. 4.1.2

[45] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the Usenix Conference on File and Storage Technologies*, Monterey, CA, January 2002. 3.1.1, 4.2.2

[46] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/4236.656066. 1

[47] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of USENIX Security Symposium*, pages 223–238, 2004. 2.3.1

[48] Henrik Sandklef, Jon-Erling Dahl, and Luis Santander. GNU Xnee. `http://http://www.sandklef.com/xnee/`. 3.1.2, 3.3.3

[49] Satyanaranyanan, M. Kozuch, M.A., Helfrich, C.J., O'Hallaron, D. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing*, 1(2):157–189, 2005. 1

[50] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andres Lagar-Cavilla. Pervasive Personal Computing in an Internet Suspend/Resume System. *IEEE Internet Computing*, 11(2), 2007. 1

[51] Bruce Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop*, pages 191–204, London, UK, 1994. Springer-Verlag. ISBN 3-540-58108-1. 3.3.3

[52] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2005. 4.1.2, 5.2.1

[53] Software Research, Inc. Testworks. `http://www.soft.com/TestWorks`. 3.1.2

[54] Technical Overview of Terminal Services. Technical Overview of Terminal Services. `http://www.microsoft.com/windowsserver2003/techinfo/overview/termserv.mspx`, January 2005. 1

[55] D. Teigland and H. Mauelshagen. Volume Managers in Linux. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001. 2.3.3

[56] Tolia, N., Harkes, J., Kozuch, M., Satyanarayanan, M. Integrating Portable and Distributed Storage. In *Proceedings of the 3rd Usenix Conference on File and Storage Technologies*, San Francisco, CA, March 2004. 4.2.2

[57] Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Bressoud, T., Perrig, A. Opportunistic Use of Content-Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, June 2003. 4.2.2

[58] Tolia, N., Satyanarayanan, M. Consistency-preserving Caching of Dynamic Database Content. In *Proceedings of the 16th International World Wide Web Conference*, Banff, Canada, May 2007. 4.2.2

[59] Tolia, N., Satyanarayanan, M., Wolbach, A. Improving Mobile Database Access over Wide-area Networks without Degrading Consistency. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, San Juan, Puerto Rico, 2007. 4.2.2

[60] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996. 4.2.2

[61] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The Personal Server: Changing the Way We Think about Ubiquitous Computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 194–209, London, UK, 2002. Springer-Verlag. ISBN 3-540-44267-7. 2.5

[62] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, San Diego, CA, June 2007. 3.2, 4.2.3

[63] Nickolai Zeldovich and Ramesh Chandra. Interactive Performance Measurement with VNCPlay. In *USENIX Annual Technical Conference, FREENIX Track*, pages 189–198, 2005. 3.1.2